VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

VIZUALIZACE ALGORITMŮ PRO PLÁNOVÁNÍ CESTY

PATH PLANNING ALGORITHMS VISUALISATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. MICHAL ŘEPKA**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. JAROSLAV ROZMAN, Ph.D.**
SUPERVISOR

BRNO 2018

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Řepka Michal, Bc.**
Obor: Inteligentní systémy
Téma: **Vizualizace algoritmů pro plánování cesty**
Path Planning Algorithms Visualisation
Kategorie: Umělá inteligence

Pokyny:

1. Nastudujte algoritmy pro plánování cesty používané v robotice (bug algoritmy, potenciálové funkce a buněčné dekompozice). Nastudujte práci Jakuba Rusnáka "Vizualizace algoritmů pro plánování cesty".
2. Do práce J. Rusnáka navrhněte sadu dalších pluginů s algoritmy vyjmenovanými výše, kterými jeho program rozšíříte. Dále navrhněte rozšíření jeho práce o možnost dávkového spouštění s různými parametry a o vypisování statistik z jednotlivých spuštění.
3. Implementujte navržené pluginy a navržené rozšíření o dávkové spouštění a statistiky. Ke každému algoritmu vytvořte přehledný popis teorie nutné k pochopení jeho funkce.
4. Vytvořte přehledovou studii, kde porovnáte statistiky běhů jednotlivých algortimů pro různé parametry a různé mapy. Výslednou práci zhodnoťte a navrhněte vylepšení.

Literatura:

- Howie Choset et al., Principles of Robot Motion, 2005, ISN 0-262-03327-5.
- Rusnák Jakub, Vizualizace algoritmů pro plánování cesty, bakalářská práce, FIT VUT v Brně, 2017.

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Rozman Jaroslav, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Nalezení bezkolizní cesty je hlavním problémem při implementaci pohyblivého, autonomního robota. Tato práce se zaměřuje na nejdůležitější algoritmy z tohoto odvětví robotiky a snaží se je srozumitelně vysvětlit. V dalších částech popisuje implementaci demonstrační aplikace, která umožňuje experimentování s těmito algoritmy. Aplikace využívá knihovnu vytvořenou pro tento účel Jakubem Rusnákem v roce 2017. Zde se tedy nachází volné pokračování a rozšíření jeho práce.

Abstract

Finding of collision free path is central in creation of mobile, autonomous robot. Goal of this paper is to show the most important algorithms implementing such solutions. It also describes application that is being created to allow students experiment with these methods. For this purpose it uses library that was introduced by Jakub Rusnák in 2017, which means this is a continuation and possibly extension of his work.

Klíčová slova

vizualizace, algoritmy plánování cesty, bug, potenciálová funkce, buněčné dekompozice, java, demonstrační aplikace

Keywords

visualisation, path planning algorithms, bug, potential function, cell decomposition, java, algorithm presentation

Citace

ŘEPKA, Michal. *Vizualizace algoritmů pro plánování cesty.* Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jaroslav Rozman, Ph.D.

Vizualizace algoritmů pro plánování cesty

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jaroslava Rozmana, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....

Michal Řepka

23. května 2018

Obsah

Kapitola 1

Úvod

Automatické plánování cesty je dobře známý problém v robotice, kde hraje důležitou roli při navigaci autonomních objektů. Ideálním cílem v tomto odvětví by bylo udat stroji ve vysoké abstrakci úlohu a moci se spolehnout, že si sám najde způsob, jak ji vyřešit pomocí svých zpětnovazebních senzorů. I když nalezení nejkratší cesty k cíli a vyhnutí se všem překážkám je typický úkol, využití algoritmů pro plánování cesty sahá mnohem dále. Dnes je autonomní navigace využívána při animaci a pohybu virtuálních objektů, plánování chirurgických zákroků, analýze montáže, mapování neznámého prostředí a dokonce návrhu léků, kde pomáhají vypočítat cesty molekul a jak se na sebe budou vázat. Plánování cesty má takto rozšířené pole působnosti, protože může být zahrnuto v jakémkoli problému, který vyžaduje výpočet sekvence kroků ke spojení počáteční a cílové konfigurace a zároveň splnění omezujících podmínek.

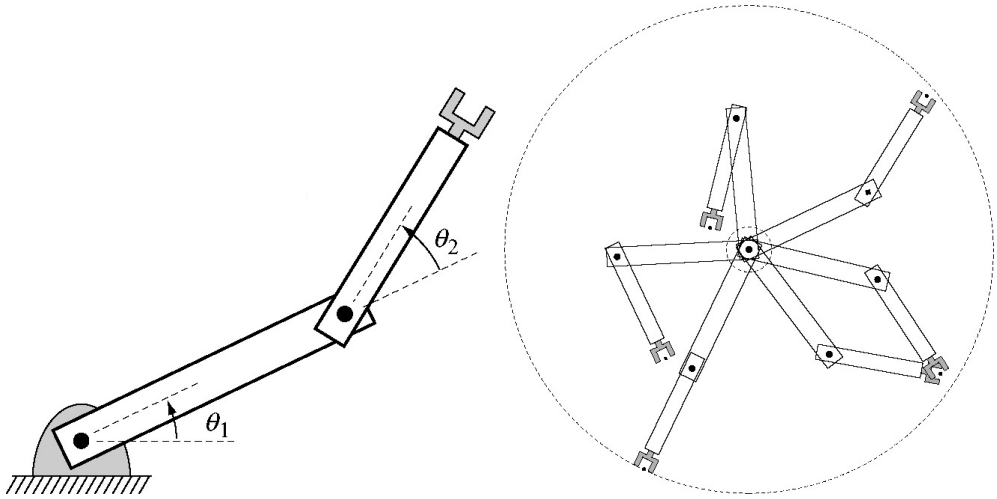
Vzhledem k tomu, že je hledání cesty tak důležitou součástí denní operace mnoha objektů, jako matematický problém byl řešen zevrubně již z mnoha úhlů pohledu. Důležitější je tedy v tuto chvíli předat tyto znalosti v co možná nejpochopitelnější formě dalším studentům, kteří na nich mohou dále stavět. Tato práce se snaží popsat a demonstrovat na příkladech jedny z nejdůležitějších algoritmů tohoto odvětví. Část z nich v podobě pravděpodobnostních a roadmap algoritmů byla popsána Jakubem Rusnákem v jeho práci [5], kde také implementoval knihovnu specializující se na vizualizaci algoritmů pro hledání cesty. Zde bude navázáno na jeho úspěšný vstup do plánovacích algoritmů. Jeho knihovna (popsána v kapitole 7.1) zde bude použita pro demonstrování funkce popisovaných přístupů (Bug algoritmy v kapitole 3, potenciálové funkce v kapitole 4 a buněčné dekompozice v kapitole 5) a v pozdějších kapitolách také rozšířena o dávkové spouštění s různými parametry a o vypisování statistik z jednotlivých spuštění pro kvantifikaci efektivity nalezených cest. Pro dosažení tohoto cíle bude potřeba mimo jiné zobecnit implementaci pluginů realizujících jednotlivé algoritmy oproti představě Jakuba Rusnáka a rozšířit o přístupnější grafické uživatelské rozhraní.

Jako primární zdroj pro tuto práci byla využita učebnice plánovacích algoritmů [2], kterou je možné doporučit při hledání dalších zdrojů informací.

Kapitola 2

Vlastnosti plánovače cesty

Klasickým problémem řešeným při plánování cesty je problém stěhování piána [6], ve kterém máme pevný, tří-dimenzionální objekt, který se snažíme dostat z bodu A do bodu B v rámci předem definovaného prostoru. V cestě stojí množina statických překážek, které omezují pozice tohoto objektu v prostoru a ovlivňují tedy i možné cesty k cíli. Abychom mohli říci, že objekt nekoliduje s nějakou překážkou, libovolný bod objektu se nesmí ocitnout v prostoru překážky. Reprezentace definující body objektu se nazývá jeho *konfigurací*. Všechny možné pozice, ve kterých se může nacházet je pak *konfigurační prostor* objektu. Příkladem reprezentace může být množina úhlů kloubů robotického ramene nebo pozice a rotace stolu. Pokud z konfiguračního prostoru odejmeme pozice, ve kterých je objekt v konfliktu s překážkou (zakázané konfigurace), pak dostáváme jeho *dostupný konfigurační prostor* (viz obrázek 2.1) [7].



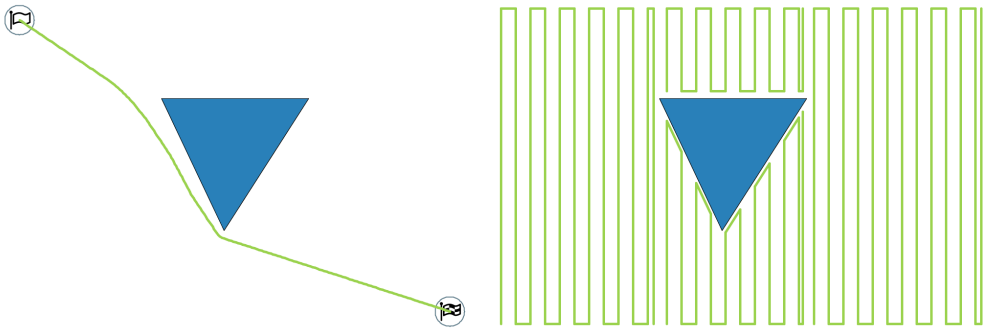
(a) (b)

Obrázek 2.1: Konfigurace dvou-kloubového ramene (2.1a) a dostupný konfigurační (pracovní) prostor otočného robotického ramene (2.1b). [2]

Vzhledem k cíli tohoto textu demonstrovat algoritmy pro hledání cesty srozumitelnou cestou, je nutné omezit prostředí, tedy i překážky a pohybující objekt samotný, do dvoudimenzí.

2.1 Účel plánování

Nejdůležitější charakterizací plánovače je typ úkolu, který plní. V následujících kapitolách budou popisovány dva z typických úkolů v robotice: *navigace* a *pokrytí*. První z nich spočívá v nalezení bezkolizní cesty z počáteční konfigurace do jiné. Příkladem aplikace takového úkolu je robotické rameno, autonomní auto nebo jiný mobilní robot. Druhým popisovaným úkolem plánovačů bude *pokrytí*. Toho je dosaženo postupným posouvání efektoru robota nad každým bodem dostupného prostoru jako například u lakování nebo odminování.



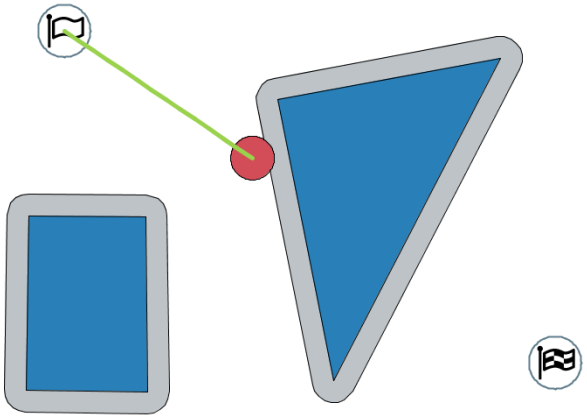
(a) (b)

Obrázek 2.2: Cesta nalezená algoritmem realizující navigaci (2.2a) a pokrytí (2.2b) nad identickým prostředím.

2.2 Vlastnosti pohyblivého objektu

Předmětem plánování cesty může být jakýkoli objekt, se kterým je potřeba bezpečně pohybovat. Plánovač je výrazně ovlivněn komplexností objektu, se kterou se zvyšuje dimenze dostupného konfiguračního prostoru. Jakmile plně rozumíme konfiguračnímu prostoru, jsme schopni naprogramovat plánovač tak, aby okamžitě určil, jestli pohyb v daném směru je možný nebo ne. Z tohoto důvodu budeme dále v textu uvažovat jednoduchého, autonomního robota, aby se čtenář mohl plně soustředit na plánovací algoritmy samotné.

Jedná se o dvou-dimenzionálního robota, jehož aktuální konfigurace je definována pozicí (x, y) a velikostí r udávající průměr kruhu, který představuje robotovu fyzickou podobu. Na obrázku 2.3 je pak možné vidět vzhled robota, dvě překážky (obdélník po levé straně a trojůhelník) a jejich rozšíření, které ohraničuje dostupný konfigurační prostor (vzhledem k velikosti robota). V průběhu popisu algoritmů budou, s případně unikátními prvky, takto vizualizovány základní útvary vyskytující se v prostředí.



Obrázek 2.3: Robot (červená), dosavadní cesta (zelená), překážky (modrá) a rozšíření překážek (šedá) udávající dostupný (bílá) a nedostupný konfigurační prostor.

2.3 Vlastnosti plánovacích algoritmů

Jakmile je definován úkol a objekt plánování, je načase vybrat algoritmus na základě jeho vlastností a toho, jak tento úkol splní. Je potřeba plánovat cestu během pohybu robota? *Online* algoritmy jsou toho schopny. Naopak *offline* algoritmy dokáží nalézt efektivnější cestu. Zde se nabízí otázka – v jakém smyslu efektivní? Je třeba s robotem co nejméně zatáčet a vydat tak co možná nejmenší množství energie? Nalézt nejkratší cestu k cíli? Ne ve všech aplikacích jsme schopni implementovat komplexní algoritmy, pak je potřeba vybrat algoritmus s menší časovou složitostí. V kritických aplikacích jsou naopak vyžadovány algoritmy *úplné*, které vždy naleznou cestu, pokud nějaká existuje, jinak v konečném čase nahlásí selhání.

Kritérií je mnoho a v tomto textu budou popisovány pouze nejvýraznější vlastnosti algoritmů, které definují jejich způsob použití. Příkladem můžou být Bug algoritmy popisované v následující kapitole, které se vyznačují použitím senzorů k pohybu v neznámém prostředí.

Kapitola 3

Bug algoritmy

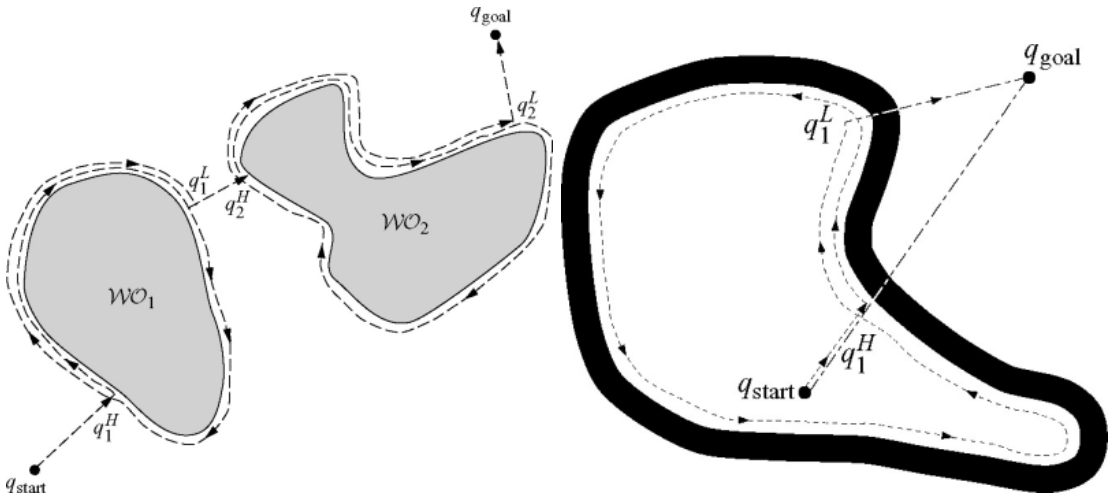
Pokud bychom chtěli najít cestu k cíli tak, jak to děláme v běžném životě, šli bychom směrem k cíli, a pokud bychom narazili na překážku, obešli bychom ji a pokračovali dále. Takto jednoduchý přístup se snaží implementovat *Bug* algoritmy, které se dále liší jen tím, v jakém bodě překážku opouští a do jaké vzdálenosti vidí.

Tato třída algoritmů je jedna z prvních a nejjednodušších způsobů hledání cesty v neznámém prostředí. Zatímco první plánovací algoritmy řešily problém stěhování piána kolem komplexních překážek v 3D prostoru, Bug algoritmy je možné definovat pouze ve dvou dimenzích. Stále to však umožňuje jejich jednoduchou matematickou definici. Tato matematická definovatelnost a dále dokazatelnost znamená, že je u nich garantováno, že pokud je cíl dosažitelný, bude také nalezen. Jedná se tedy o úplný algoritmus, což je velmi důležitá vlastnost v tomto oboru.

3.1 Bug 1

Robot je zde brán jako jednoduchý bod na ploše s počáteční a cílovou pozicí a překážkami označenými svým obvodem. Je vybaven dotykovým senzorem, který mu umožňuje se libovolně pohybovat a při nalezení překážky se neponičit. Dále je schopen se pohybovat po obvodu jakékoli překážky. Vždy zná svoji pozici a je pomocí ní schopen vypočítat svoji aktuální vzdálenost od cíle a směr, který ho k němu dovede.

Na obrázku 3.1 je vidět, jak tento algoritmus postupuje. Počáteční a cílová pozice je zde označena jako q_{start} a q_{goal}. Na začátku můžeme označit $q_0^L \leftarrow q_{start}$ a robot se tak pohybuje po m-přímce spojující vždy q_i^L s cílem. Parametr i značí, ke kolikáté nalezené překážce bod patří; na počátku přiřadíme $i \leftarrow 0$. Pokud bychom v tuto chvíli nenarazili na překážku, triviálně bychom došli do q_{goal}. S detekovanou překážkou si danou pozici uložíme jako q_1^H, H podle angl. *hit point* a L pro angl. *leave point*. Robot nyní obejde celý obvod překážky a znovu se tedy ocitá v bodě q_1^H. Jelikož si pamatuje cestu, kterou prošel kolem aktuální překážky, může si vypočítat q_1^L jako bod na této cestě (obvodu překážky), který je nejblíže k cíli. Přejde do tohoto bodu a ocitá se v téměř stejné situaci jako na počátku algoritmu, z důvodu speciálního případu, kdy je obklíčen jednou uzavřenou překážkou a cíl je mimo ni, jak je možné vidět na obrázku 3.1. Pokud by tedy z bodu q_1^L byl cíl stejným směrem jako právě překročená překážka, jedná se o tento případ a algoritmus neúspěšně končí. Jinak se vrací na začátek a pokračuje po nově aktualizované m-přímce.



(a) (b)

Obrázek 3.1: Algoritmus Bug 1 nalézá cíl po překročení dvou překážek (3.1a) a zjišťuje, že cíl je nedostupný (3.1b). [2]

Algoritmus již na první pohled není příliš efektivní vzhledem k tomu, že konečná cesta obsahuje navíc délku obvodů všech nalezených překážek. Tato metoda ovšem jednoduše zaručuje úplnost tohoto algoritmu, pokud se můžeme spolehnout, že všechny překážky jsou uzavřeny.

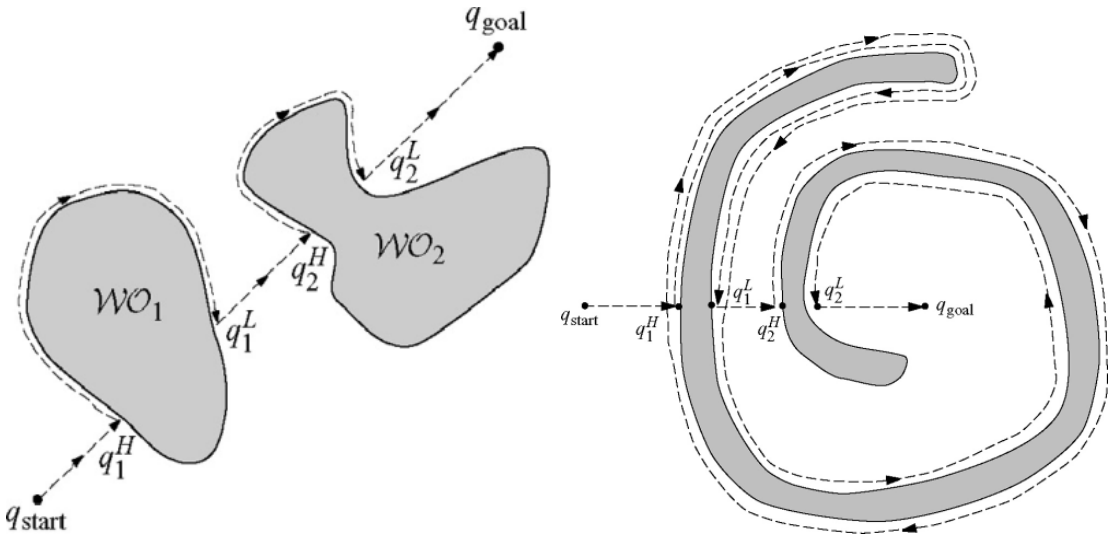
Algoritmus 1: Bug 1

Input: Robot o velikosti bodu s dotykovým senzorem.
Output: Cesta do cíle q_{goal} nebo závěr, že cesta neexistuje.

while *true* **do**
  repeat
    pohybuj se z bodu q_{i-1}^L směrem k q_{goal}
  until q_{goal} *je dosažen* ***or*** *překážka je nalezena v bodě* q_i^H
  if q_{goal} *je dosažen* **then**
    return *cestu k* q_{goal}
  end if
  repeat
    pohybuj se po obvodu překážky
  until q_{goal} *je dosažen* ***or*** q_i^H *je znovu dosažen*
  vypočítej bod q_i^L na obvodu s nejkratší vzdáleností k q_{goal}
  jdi do bodu q_i^L
  if *robot by ve směru cíle narazil na stejnou překážku* **then**
    return *cesta neexistuje*
  end if
  inkrementuj i
end while

3.2 Bug 2

Kolega předcházejícího algoritmu označovaný Bug 2 využívá stejného modelu robota a senzoru. Stejně jako on využívá dvě hlavní fáze: pohyb směrem k cíli a pohyb po obvodu překážky. Obě fáze jsou ovšem lehce modifikovány, což má za cíl nalezení ve většině případech až násobně kratší cesty k cíli. Hlavní příčinou této modifikace je jiná definice m-přímky, která je nyní stacionární spojnicí mezi q_{start} a q_{goal}.



(a) (b)

Obrázek 3.2: Algoritmus Bug 2 nalézá cíl po překročení dvou překážek (3.2a) a extrémní případ (3.2b) [2]

Zprvu je tato přímka stejná, robot se po ní může pohybovat směrem k cíli a po nalezení překážky zapíše bod q_i^H. Poté se stejně jako u algoritmu Bug 1 pohybuje po obvodu překážky, aby nalezl její druhou stranu – bod q_i^L, který je jeho dalším setkáním s m-přímkou, tentokrát bližším k cíli než minule. Nyní může robot pokračovat směrem k cíli a opakovat cyklus. Takto úspěšný běh je možné vidět na obrázku 3.2a. V pravé části obrázku 3.2 je pak případ, kvůli kterému hledáme *bližší* (řádek 8.3 v algoritmu 2) průsečík cesty robota s m-přímkou. Pokud bychom se spokojili s jakýmkoli průsečíkem, narazili bychom opět na q_1^H a algoritmus by se zacyklil.

Algoritmus 2: Bug 2

Input: Robot o velikosti bodu s dotykovým senzorem.
Output: Cesta do cíle q_{goal} nebo závěr, že cesta neexistuje.

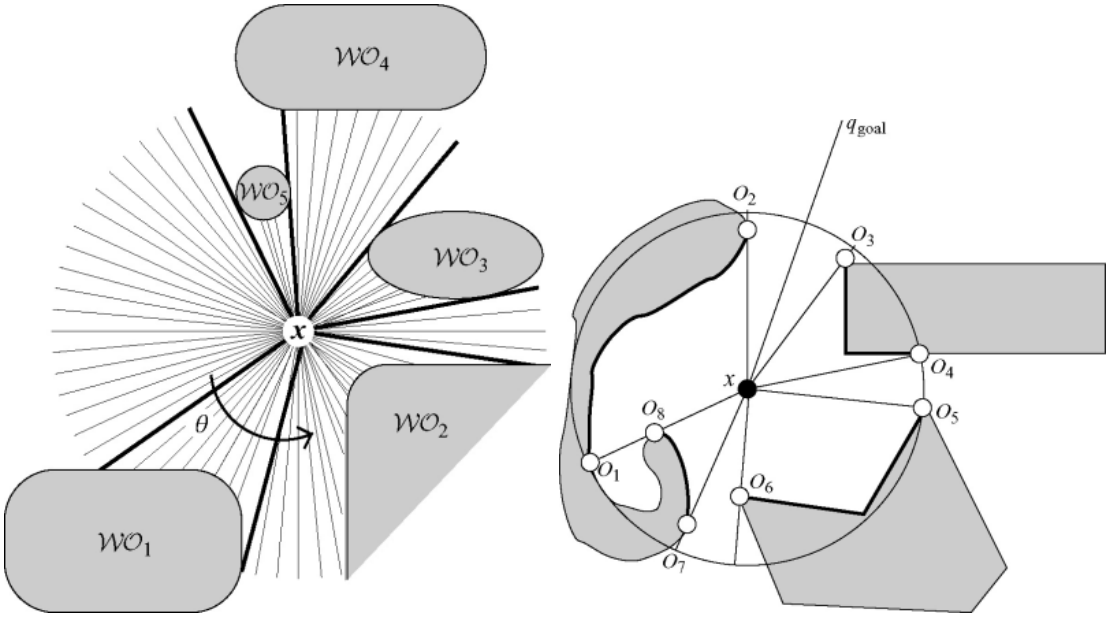
while *true* **do**
  repeat
    pohybuj se z bodu q_{i-1}^L směrem k q_{goal} po m-přímce
  until q_{goal} *je dosažen* ***or*** *překážka je nalezena v bodě* q_i^H
  if q_{goal} *je dosažen* **then**
    return *cestu k* q_{goal}
  end if
  otoč se doleva (nebo doprava)
  repeat
    pohybuj se po obvodu překážky
  until q_{goal} *je dosažen* ***or***
  q_i^H *je znovu dosažen* ***or***
  m-přímka je znovu dosažena v bodě $m \neq q_i^H$, $d(m, q_{goal}) < d(q_i^H, q_{goal})$
  if *robot by ve směru cíle narazil na stejnou překážku* **then**
    return *cesta neexistuje*
  end if
end while

Délka nalezené cesty algoritmu Bug 2 se může zdát na první pohled jasně kratší než u Bug 1. Není tomu tak ovšem vždy. Tento algoritmus totiž může jednu překážku obcházet až n-krát, kde n je počet průsečíků m-přímky s překážkou, jak je ukázáno na obrázku 3.2b a výrazně si tak protáhnout výslednou cestu. Komplexnější překážky tedy mohou způsobit, že první z těchto dvou algoritmů bude lepším a jednodušším řešením.

3.3 Tangent Bug

Algoritmus Tanget Bug pracuje s robotem jako bodem na dvourozměrné doméně stejně jako v předešlých případech, tentokrát je však vybaven senzorem s libovolným dosahem. To mu umožňuje nalézt kratší cestu, než jakou je schopen nalézt Bug 2.

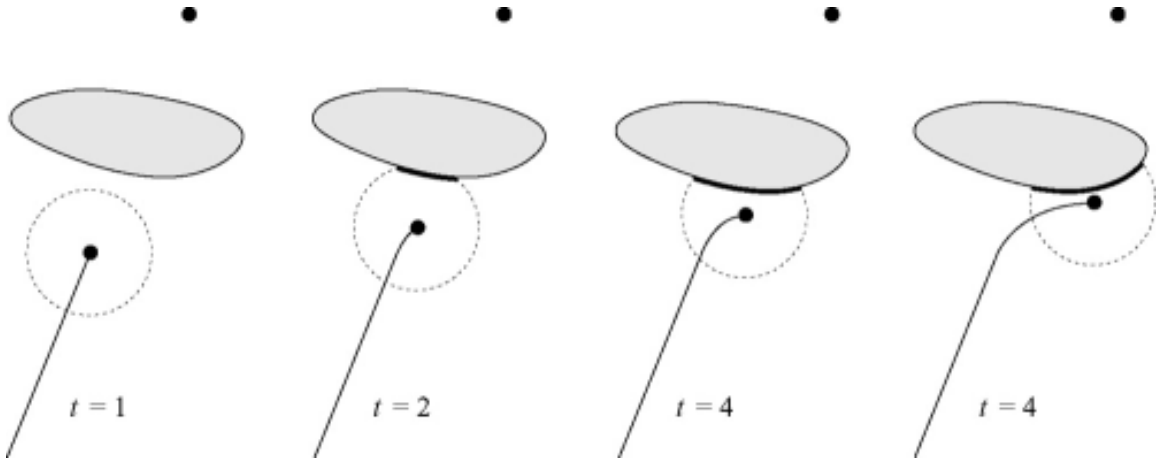
Předpokládá se, že robot je schopen nalézt nespojitosti z výstupu svého senzoru. Spojité intervaly odpovídají vzdálenostem k jednotlivým překážkám a paprsky, které neprotnou žádnou překážku nabývají hodnoty nekonečno. Na obrázku 3.3 vlevo jsou tyto paprsky označeny tence a končí v prostoru, silně označené paprsky jsou změny ve spojitosti způsobené buď ojedinělou překážkou nebo jednou překážkou blokující druhou. Takto nalezené body nespojitosti budou značeny O_n.



(a) (b)

Obrázek 3.3: Senzor robota v Tangent Bug algoritmu se zvýrazněnými nespojitostmi. [2]

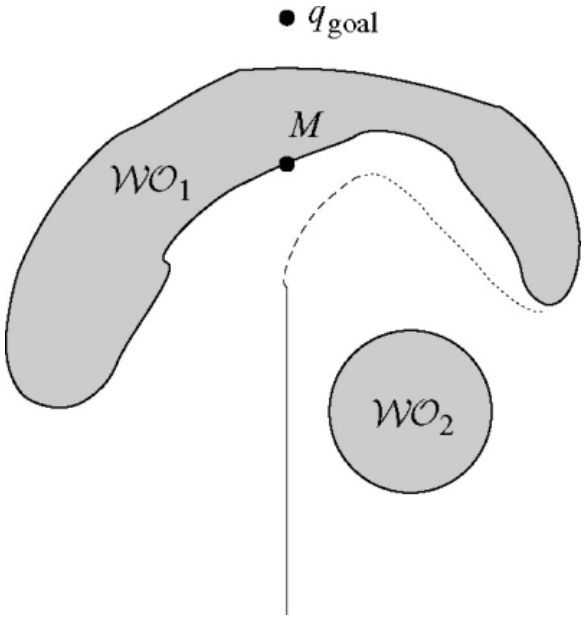
Stejně jako předešlé algoritmy Bug i Tangent Bug přechází při hledání cíle mezi dvěma módy pohybu. Oba jsou však výrazně odlišné. Pohyb k cíli sice funguje stejně bez překážek v cestě. Jakmile na horizontu senzoru spatří překážku, aktualizuje O_i a tak se jí začne postupně vyhýbat, až se eventuálně pohybuje paralelně k ní. Toto chování je znázorněno na obrázku 3.4.



Obrázek 3.4: Robot ve fázi pohybu k cíli algoritmu Tangent Bug. [2]

Od tohoto módu pohybu se robot odkloní až ve chvíli, kdy se ocitne v lokálním minimu. Tuto skutečnost zjistí ve chvíli, kdy se aktuálním krokem vzdálí od cíle, místo toho, aby se k němu přiblížil. Aby věděl, kdy z módu pohybu po obvodu přepnout zpět, uloží si vzdálenost $d_{followed}$ mezi bodem M na povrchu obcházené překážky, který je nejblíže k cíli. Nalezení bodu M je znázorněno na obrázku 3.5. Dále pokračuje v módu pohybu po obvodu překážky ve stejném směru, v jakém se pohyboval doposud. S dalšími lokálními minimy

aktualizuje hodnotu $d_{followed}$. Mezitím v každém kroku aktualizuje hodnotu d_{reach}, která odpovídá vzdálenosti mezi nejbližším bodem na obvodu obcházené překážky a cílem. Robot se odkloní od překážky a pokračuje směrem k cíli ve chvíli, kdy $d_{reach} < d_{followed}$.



Obrázek 3.5: Pohyb před nalezením překážky plnou čarou, dále pohyb k bodu O_i čárkovaně, poté uložení bodu M a pohyb podél obvodu tečkovaně. [2]

Cesta k cíli nalezená Tangent bugem nabývá různých délek a dokonce i různých tras v závislosti na dosahu senzoru robota. I s nulovým dosahem nalézá tento algoritmus výrazně lepší cesty než Bug 1 a 2. Čím vyšší rozsah, tím lepší rozhodnutí je robot schopen dělat ve chvíli, kdy se ocitá před překážkou. S dosahem blížícím se nekonečnu pak robot obchází všechna lokální minima velice podobně jako loď na moři míjí všechny zátoky. Taková cesta je optimální cestou pro dané prostředí.

Algoritmus 3: TANGENT BUG

Input: Robot o velikosti bodu se senzorem vzdálenosti od překážek.
Output: Cesta do cíle q_{goal} nebo závěr, že cesta neexistuje.

1: **while** *true* **do**
2: **repeat**
3: aktualizuj hodnoty O_i
4: pohybuj se směrem k bodu $n \in O_i$, který minimalizuje $d(x, n) + d(n, q_{goal})$
5: **until** q_{goal} *je dosažen* ***or***
směr, který minimalizuje $d(x, n) + d(n, q_{goal})$ *začne zvyšovat* $d(x, q_{goal})$
6: **if** q_{goal} *je dosažen* **then**
7: **return** *cestu k* q_{goal}
8: **else**
9: robot se ocitl v lokálním minimu
10: **end if**
11: ulož směr s, ve kterém ses kolem této překážky pohyboval doteď
12: **repeat**
13: aktualizuj hodnoty d_{reach}, $d_{followed}$ a O_i
14: pohybuj se směrem s k bodu $n \in O_i$
15: **until** q_{goal} *je dosažen* ***or***
robot dokončí okruh kolem překážky, takže q_{goal} *je nedosažitelný* ***or***
$d_{reach} < d_{followed}$
16: **if** q_{goal} *je dosažen* **then**
17: **return** *cestu k* q_{goal}
18: **else**
19: **return** *cesta neexistuje*
20: **end if**
21: **end while**

Kapitola 4

Přitažlivé a odpudivé potenciálové funkce

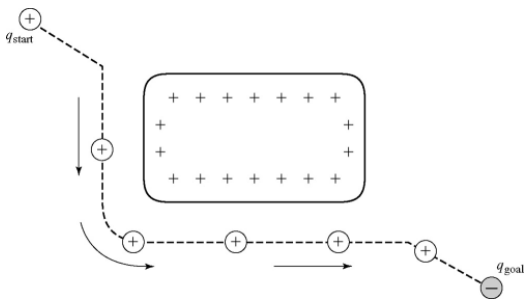
Minulá kapitola se zabývala algoritmy, které nevyžadovaly explicitní reprezentaci konfiguračního prostoru. Jediné, co bylo třeba definovat, byl postup, jak robot postupně objevuje své okolí a reaguje na překážky. I když je to přístup jednoduchý a zřejmě účinný, se znalostí konfiguračního prostoru a jeho vhodnou interpretací, je třída algoritmů, popisovaná v této kapitole, schopna generovat mnohem obecnější řešení. Díky své obecnosti je možné tyto algoritmy dále rozšířit do mnohem komplexnějších konfiguračních prostorů.

Potenciál

Samotný dostupný prostor je v této kapitole reprezentován jako vektorové pole, které udává robotu v každém kroku směr k cíli. Potenciálem vektorového pole $F : \mathbb{R}^n \to \mathbb{R}^n$ nazveme takovou reálnou funkci $U : \mathbb{R}^n \to \mathbb{R}$, která splňuje

$$F_i = \frac{\partial U}{\partial q_i}.$$

Vektorové pole, které má potenciál, můžeme nazvat potenciálové. Jelikož hodnota potenciálové funkce může být brána jako energie, gradient potenciálu je pak vektor síly (s jistým zjednodušením fyzikálních vlivů pouze vektor rychlosti). Tento gradient můžeme popsat pomocí vektoru $\Delta U(q) = DU(q)^T = [\frac{\partial U}{\partial q_1}(q), \ldots, \frac{\partial U}{\partial q_n}(q)]^T$, který nabývá směr lokálně maximalizující U. Pokud ho znegujeme, dostáváme nástroj pro tzv. gradientní sestup a nalezení minima.



Obrázek 4.1: Kladně nabitá překážka robota odpuzuje zatímco záporně nabitý cíl jej přitahuje.[2]

Přístup s potenciálovou funkcí je schopen směřovat robota jako částici ve vektorovém poli gradientu (obrázek 4.1). Tento gradient pak působí na kladně nabitého robota určitou silou, která ho směřuje k záporně nabitému cíli. V dalších částech této kapitoly budou popsány síly, které robota odpuzují od překážek. Ty tedy díky analogii z magnetického pole můžeme označit jako kladně nabité.

Nejjednodušší potenciálovou funkcí je přitažlivá/odpudivá (*attractive/repulsive*) potenciálová funkce, která funguje pomocí dvou efektů: cíl robota přitahuje, zatímco překážky jej odpuzují. Suma těchto efektů umožňuje přitáhnout robota do cíle a přitom se vyhnout překážkám:

$$U(q) = U_{att}(q) + U_{rep}(q).$$

4.1 Přitažlivý potenciál

Tato funkce by měla splňovat několik kritérií:

- monotónní, rostoucí funkce
- spojitě diferencovatelná
- s větší vzdálenosti od cíle (rozumně) větší hodnotu a s menší naopak

Pro splnění prvního kritéria je nejjednodušší vybrat *kuželovitý potenciál*, který měří vzdálenost od bodu q do q_{goal}:

$$U_{att}(q) = \zeta d(q, q_{goal}),$$

$$\Delta U_{att}(q) = \frac{\zeta}{d(q, q_{goal})}(q - q_{goal}),$$

kde ζ je parametr pro ovlivnění efektu přitažlivosti. Druhá rovnice je gradient odvozený od tohoto potenciálu. Nyní s počátkem v jakémkoli bodě je možné nalézt cestu do cíle sledováním znegovaného gradientu této funkce. Nedefinovaná je pouze v jednom bodě – cíli. Druhé kritérium je zavedeno z důvodu kmitání hodnoty funkce kolem nespojitostí, v tuto chvíli reprezentovaných jen cílem. Abychom se vyhnuli tomuto problému, potřebujeme funkci, která bude mít výrazně menší hodnotu, jakmile se q bude přibližovat k cíli:

$$U_{att}(q) = \frac{1}{2}\zeta d^2(q, q_{goal}),$$

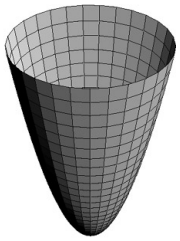
$$\Delta U_{att}(q) = \zeta(q - q_{goal}).$$

Opět nejjednodušším řešením je funkce rostoucí kvadraticky se vzdáleností od q_{goal}. S vyřešenými malými hodnotami zbývá ještě problém s neovladatelným růstem funkce se vzdalujícím se q. Zkombinujeme tedy kvadratický a kuželovitý potenciál do jedné funkce s vhodně zvoleným přechodem ve vzdálenosti d^*_{goal}:

$$U_{att}(q) = \begin{cases} \frac{1}{2}\zeta d^2(q, q_{goal}), & d(q, q_{goal}) \leq d^*_{goal} \\ d^*_{goal}\zeta d(q, q_{goal}) - \frac{1}{2}\zeta (d^*_{goal})^2, & d(q, q_{goal}) > d^*_{goal} \end{cases},$$

$$\Delta U_{att}(q) = \begin{cases} \zeta(q - q_{goal}), & d(q, q_{goal}) \leq d^*_{goal} \\ \frac{d^*_{goal}\zeta(q - q_{goal})}{d(q, q_{goal})}, & d(q, q_{goal}) > d^*_{goal} \end{cases}.$$

Jelikož je funkce definovaná i na přechodu, nevznikají zde žádné nové nespojitosti.



Obrázek 4.2: Graf atraktivního potenciálu.[2].

4.2 Odpudivý potenciál

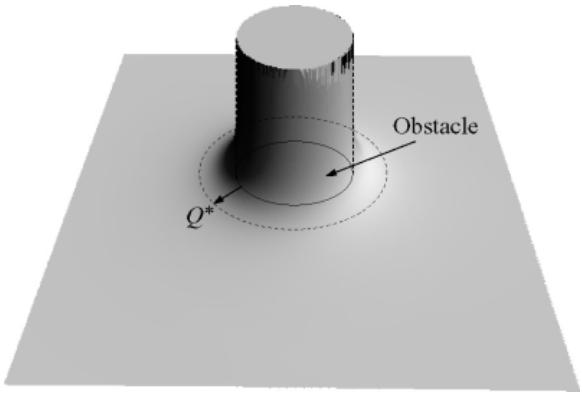
Odpudivý potenciál drží robota v bezpečné vzdálenosti od překážek, jeho síla tedy musí záviset na vzdálenosti od nejbližší z nich $D(q)$:

$$U_{rep}(q) = \begin{cases} \frac{1}{2}\eta(\frac{1}{D(q)} - \frac{1}{Q^*})^2, & D(q) \leq Q^* \\ 0, & D(q) > Q^* \end{cases}.$$

Gradient je pak:

$$\Delta U_{rep}(q) = \begin{cases} \eta(\frac{1}{Q^*} - \frac{1}{D(q)})\frac{1}{D^2(q)}\Delta D(q), & D(q) \leq Q^* \\ 0, & D(q) > Q^* \end{cases}.$$

Hodnota $Q^* \in \mathbb{R}$ umožňuje robotovi ignorovat vliv překážek, které jsou od něj v dostatečné vzdálenosti a parametr η zvyšuje odpudivou sílu s přibližující se překážkou.



Obrázek 4.3: Odpudivý gradient má vliv pouze v blízkosti překážky.[2].

4.3 Gradientní sestup

Sestup gradientu je často používaná metoda u problémů optimalizace. V počáteční konfiguraci stačí udělat malý krok proti směru gradientu, čímž se robot dostane do nové konfigurace a proces se může opakovat.

Algoritmus 4: Sestup gradientu

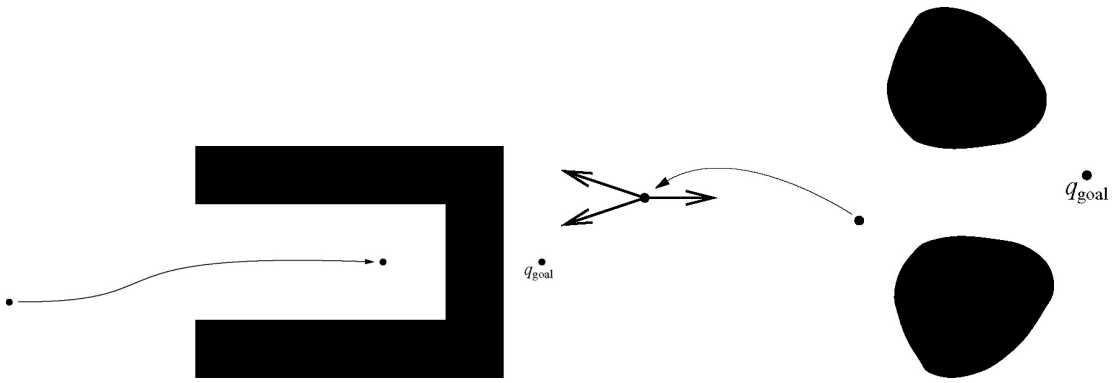
Input: Schopnost vypočítat gradient $\Delta U(q)$ v bodě q.
Output: Sekvence bodů $\{q_0, q_1, \ldots, q_i\}$.

1: $q_0 = q_{start}$
2: $i = 0$
3: **while** $\Delta U(q_i) \neq 0$ **do**
4: $\quad q_{i+1} = q_i + a_i \Delta U(q_i)$
5: $\quad i = i + 1$
6: **end while**

V algoritmu 4 je vytvořena cesta postupnou iterací nad q_{start} jako počátečním bodem. Hodnota a_i udává velikost kroku v i-té iteraci, která musí být dostatečně malá, aby nebylo dovoleno robotu *vkročit* do překážky a dostatečně velká, aby nebyl výpočet zbytečně náročný. Stejně tak ukončovací podmínka bývá ve skutečnosti volnější v tom, že není potřeba dojít přesně do $\Delta U(q_i) = 0$ (není to většinou ani možné), ale je zvolena malá konstanta ϵ, s jejíž vzdáleností od cíle je robot spokojen.

4.4 Problém lokálního minima

Problém, který je vždy spojen s metodou sestupu gradientu, je problém lokálního minima. Je dokazatelné, že se správně zvolenými hodnotami a_i, robot sestoupí do minima, není však jisté, že se jedná o minimum globální (cíl). Na obrázku 4.4a je vidět lokální minimum, do kterého se robot dostane, protože je přitahován k cíli. Ve chvíli, kdy je v bodě před překážkou a měl by ji obejít, netuší, že na konci překážky ve tvaru *podkovy* není průchod. Podobný osud může nastat, pokud se robot dostane do lokálního minima ve tvaru horského hřebenu vytvořeného dvěma překážkami ve stejné vzdálenosti zobrazeného na obrázku 4.4b vpravo.



(a) (b)

Obrázek 4.4: Lokální minimum v uzavřené překážce (4.4a) a mezi dvěma překážkami (4.4b).[2]

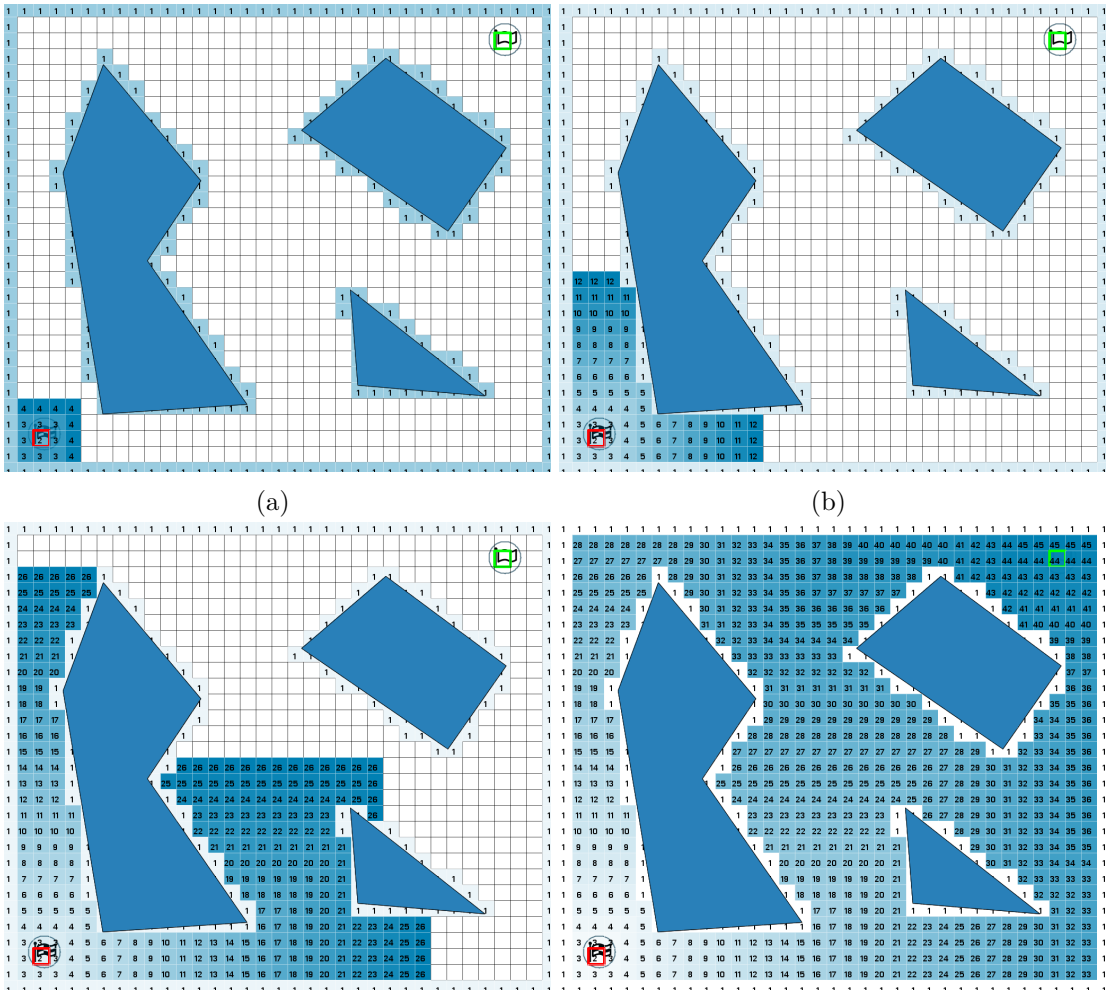
Tomuto problému lze čelit algoritmy s velkou mírou náhodnosti, které se snaží buď lokální minima obejít nebo se z nich dostat náhodnými průchody. Zde však upravíme výpočet vzdálenosti $d(q, q_{goal})$ tak, aby bral v úvahu i překážky stojící v cestě a nejen vzdálenost *leteckou*.

4.5 Wavefront algoritmus

Tento algoritmus implementuje vzdálenost mezi bodem q a cílem a bere přitom v úvahu pouze realizovatelné cesty. Pracuje však pouze na prostoru rozděleném do mřížky, která je vidět na příkladu na obrázku 4.6. U výpočtu vzdálenosti na mřížce se nabízí otázka, jakou formu okolí zvolit; nabízí se čtyř a osmi-okolí (na obrázku 4.5) a závisí na výběru uživatele nebo schopnostech robota traverzovat diagonálně.

(a) (b)

Obrázek 4.5: Čtyř-okolí (4.5a) a osmi-okolí (4.5b) pole označeného 2.



(a) (b)

(c) (d)

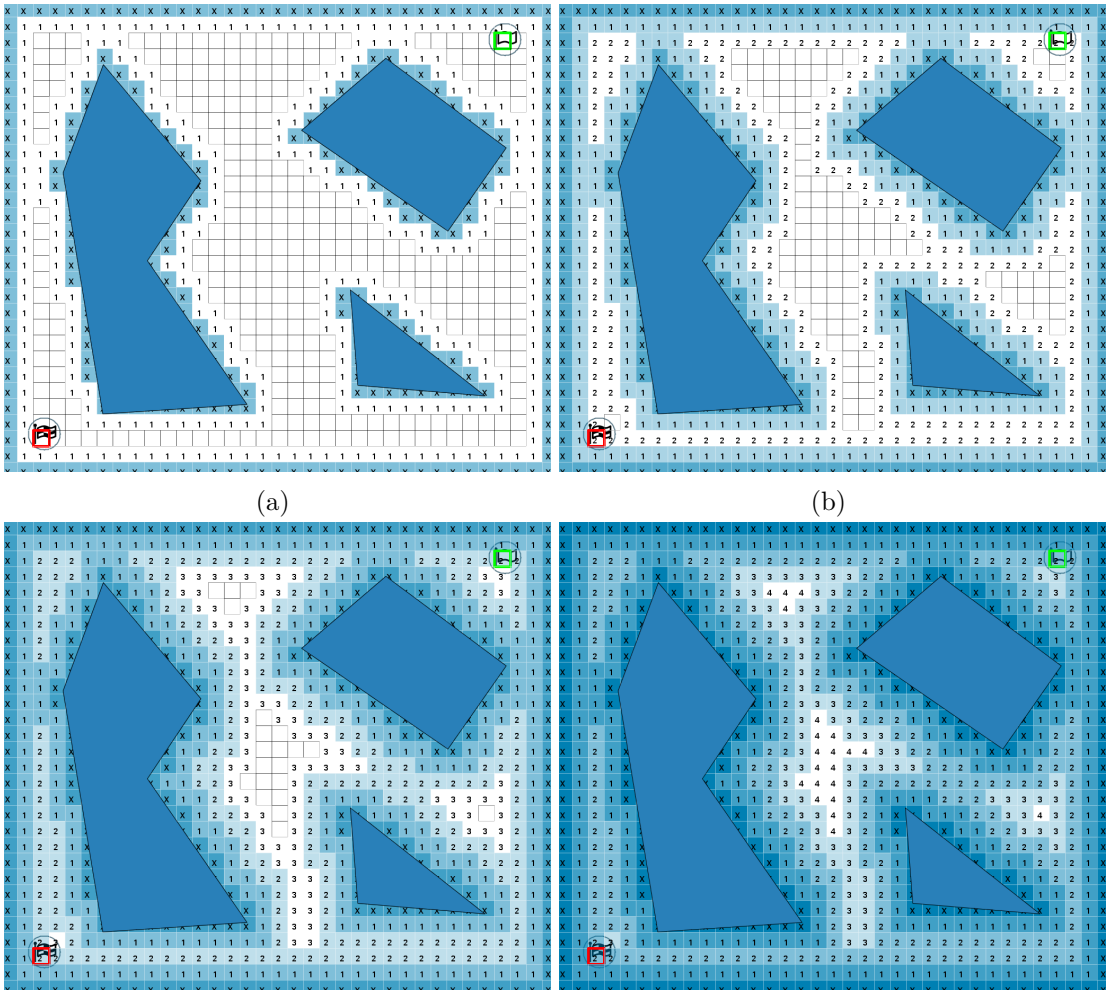
Obrázek 4.6: Propagace vlny Wavefront algoritmu při výpočtu vzdálenosti od cíle.

Algoritmus začíná s arbitrárním označením nuly pro volný prostor a jedničkou pro pole, které obsahují jakkoli velký kus překážky. Startovní pozice je označena na obrázku zeleně a cílová červeně, obě důležitá pole algoritmus zná. Cílové pole je označeno první hodnotou vzdálenosti – dva. Nyní jsou vybrány sousední pole cíle, která ještě nebyla ohodnocena, měla tedy prozatím hodnotu nula a jsou nově označeny trojkou. Pokračuje se s nulou označenými sousedy těchto polí, které získávají hodnotu čtyři a tak dále, dokud je možné nalézt ještě nějaké sousedy s nulou. Ve skutečnosti není potřeba počítat pole ve chvíli, kdy je už ohodnocené pole startu, protože taková pole nemohou být součástí nejkratší cesty. Zde je výpočet dokončen pro ilustrační účely a hodnoty pole s překážkou nastaveny na maximum, aby odpovídaly tmavým, nedostupným polím.

4.6 Brushfire algoritmus

Stejně jako výpočet vzdálenosti u přitažlivé potenciálové funkce, i odpudivá funkce vyžaduje výpočet vzdálenosti od překážek na mřížce, aby bylo možné vytvořit sumu potenciálů a celkové diskretizované řešení.

Opět se začíná s arbitrárním označením polí: nula pro volné pole, jedna pro pole s překážkou. V prvním kroku se vezmou všechna volná, nulou ohodnocená pole sousedící s poli s jedničkou (překážkou) a získají hodnotu dva. Pokračuje se stejně jako u algoritmu Wavefront s dalšími poli, dokud nejsou všechna ohodnocena, jak je vidět na obrázku 4.7.



(a) (b)

(c) (d)

Obrázek 4.7: Propagace výpočtu vzdálenosti jednotlivých polí od nejbližších překážek.

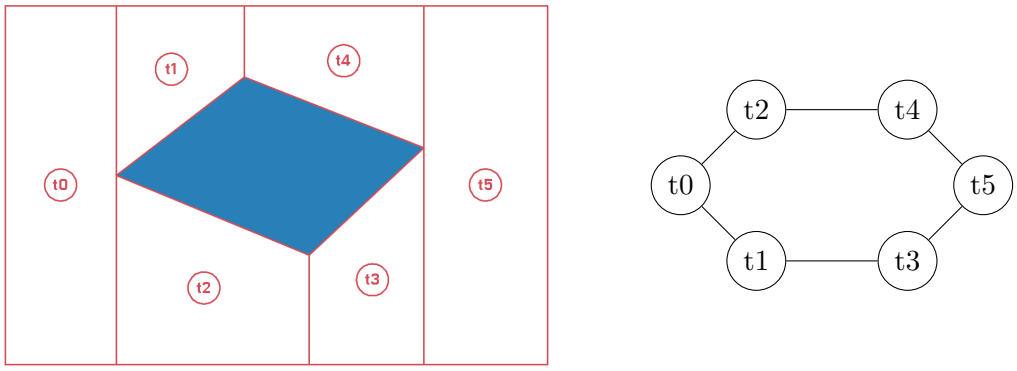
Algoritmus Brushfire generuje mřížku hodnot vzdáleností od nejbližší překážky. Tyto hodnoty mohou být použity k výpočtu odpudivého potenciálu a gradient je možné v každém bodě získat jako směr k sousednímu poli s nejnižší hodnotou.

Kapitola 5

Buněčné dekompozice

Schopnost diskretizace reprezentace konfiguračního prostoru je u algoritmů hledání cesty žádanou vlastností. Umožňuje jim efektivně a především exaktně určit trasu známým prostředím. V minulé kapitole byl prostor rozdělen do jemné mřížky, která pouze aproximovala vzhled původních překážek. Nyní uvažujme reprezentaci pomocí exaktní dekompozice do buněk. Dekompozice je exaktní, pokud sjednocením všech buněk dostáváme právě celý dostupný konfigurační prostor. Zmiňovaná mřížka tuto podmínku nesplňovala vzhledem k nedostupným buňkám v blízkosti původních překážek.

Sdílené hrany mezi buňkami nejsou globálně definované, ale záleží na lokálních podmínkách jako je změna nejbližší nebo sousedící překážky. Řekněme, že dvě buňky sousedí, pokud sdílejí takovou hranu. Nad dekompozicí pak můžeme definovat neorientovaný graf sousednosti, který reprezentuje tyto vztahy nad všemi buňkami v prostoru. Uzly tohoto grafu jsou samotné buňky, které jsou spojeny hranou právě tehdy, pokud spolu sousedí (viz 5.1).



(a) (b)

Obrázek 5.1: Sousedící buňky 5.1a a odpovídající graf sousednosti 5.1b.

S nalezenou dekompozicí dostupného prostoru je možné naplánovat cestu mezi počáteční a cílovou konfigurací ve dvou krocích. Nejprve jsou určeny buňky odpovídající startu a cíli. Poté je hledána cesta mezi těmito buňkami v grafu sousednosti pomocí grafových algoritmů jako je například A* [3]. Cesta skrze buňky samotné je vzhledem k jejich primitivnímu tvaru často triviální.

Buněčné dekompozice však nemusí být použity jen k hledání cesty do cíle. Vyznačují se tím, že umožňují vytvářet i cesty pokrývající celý dostupný prostor (coverage). Namísto

cesty v grafu sousednosti z počátečního do cílového uzlu je nalezen průchod grafem, obsahující všechny uzly (předpokládejme souvislý graf). Cesta skrze buňky pak není jen od jedné hrany k druhé, ale pokrývá ji celou, například pohybem nahoru a dolů podobně jako při orání pole.

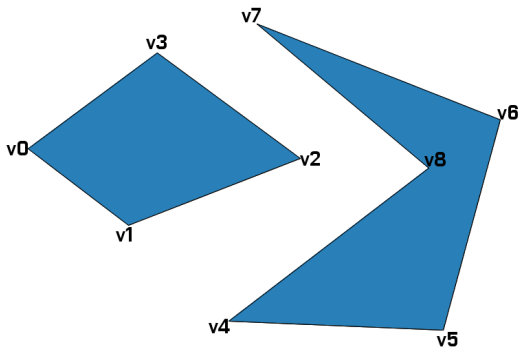
5.1 Lichoběžníková dekompozice

Nejpoužívanější dekompozicí je vzhledem k její jednoduchosti dekompozice vytvářející buňky ve tvaru lichoběžníků. Dalo by se říci, že rozdělení na samotné trojúhelníky je v této oblasti přirozenější. To tak ovšem není a dokonce mnohé algoritmy na rovinnou triangulaci[1] používají lichoběžníky jako svou počáteční dekompozici. Důvod by měl být zřejmý po pochopení konstrukce lichoběžníkových buněk.

Tato dekompozice je omezená na polygonální prostředí, tedy prostředí, ve kterém je každý objekt, včetně vnější hranice dostupného prostoru, polygonem. Algoritmy schopné rozdělit libovolné prostředí jsou popsány v pozdějších sekcích počínaje 5.3 Morseovými dekompozicemi.

Definice dekompozice

Jak již bylo řečeno, cílem tohoto algoritmu je rozdělit prostor na množinu dvou-dimenzionálních buněk ve tvaru lichoběžníků. V některých případech jsou tyto buňky degenerovanými lichoběžníky s jednou stranou o délce nula – trojúhelníky. Uvažujme, že polygonální prostor je definován množinou vrcholů překážek a vnějších hranic v_i na pozicích (x, y), kde každý vrchol má unikátní x-ové souřadnice, tedy $\forall i \neq j : v_{i_x} \neq v_{j_x}$.

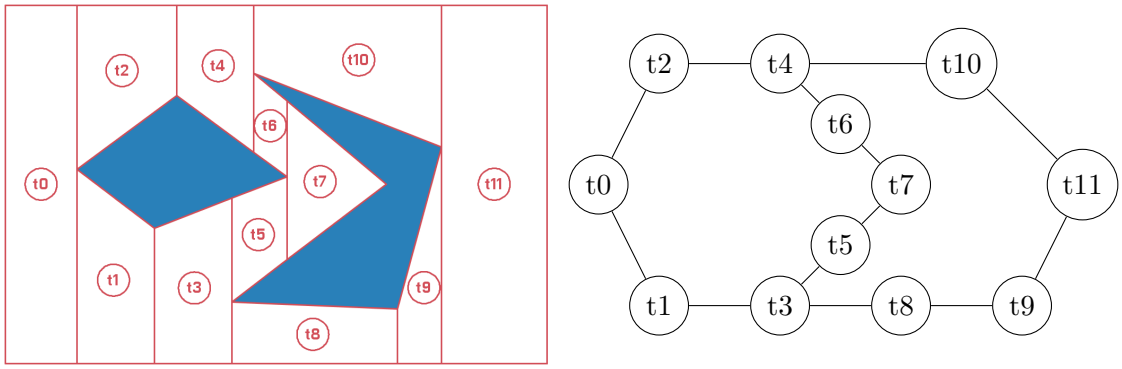


Obrázek 5.2: Polygonální prostor definovaný vrcholy v_i.

Pro vytvoření dekompozice umístíme dvě polopřímky do každého vrcholu. Jednu směrem nahoru a druhou směrem dolů, tedy do obou stran osy y. Tyto polopřímky ukončíme a vytvoříme tak vertikální úsečky ve chvíli, kdy narazí na první hranu nějaké překážky. Pro mnohé vrcholy to znamená, že vygenerují úsečku pouze jedním směrem (vrcholy v_1 a v_3 na obrázku 5.2), jiné nevygenerují ani jednu (vrchol v_8 na obrázku 5.2).

[1] Triangulace je planární rozdělení prostoru na trojúhelníky.

Dvojice následných vertikálních úseček v x-ové ose nyní vytváří společně s hranami překážek lichoběžníky a tedy dekompozici prostředí. Pokud aplikujeme pravidlo o sousedství na sdílené hrany buněk, jsme schopni získat graf sousednosti vytvořené dekompozice.

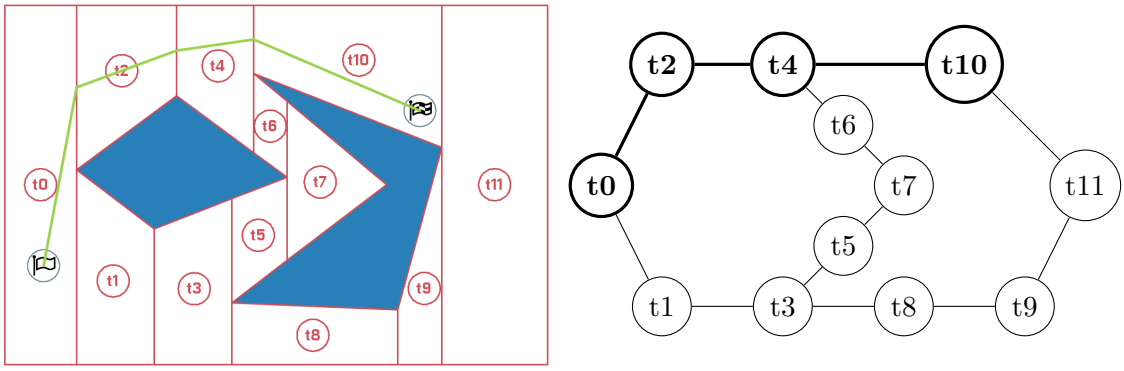


(a) (b)

Obrázek 5.3: Dekompozice prostoru na lichoběžníky 5.3a a graf sousednosti 5.3b.

Jakmile určíme, ve kterých buňkách se nachází počáteční a cílová konfigurace, nalezení cesty v grafu je přímočaré. Problém je v tom, že tato cesta nám přímo neříká, jakými konfiguracemi prostředí projít, abych se vyhnuli překážkám.

Lichoběžník je konvexní geometrický útvar. Má tedy tu vlastnost, že jakékoli dva body na jeho hranách mohou být propojeny jím plně obsaženou úsečkou. Pro jednoduchost si vyberme na sousedících hranách buňky střed úsečky a tyto body spojme. Pokud tak budeme postupovat po buňkách definovaných cestou v grafu a nakonec propojíme i start a cíl, získáme tak naši cestu prostředím.



(a) (b)

Obrázek 5.4: Cesta propojující středy společných hran lichoběžníků 5.4a, cesta v grafu sousednosti buněk 5.4b.

Konstrukce dekompozice

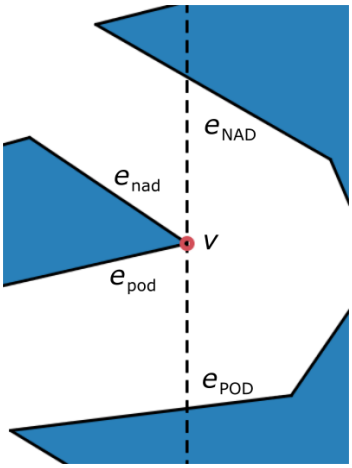
Vstupem algoritmu je list polygonů, každý reprezentován listem vrcholů. Prvním krokem je získat výše definované vertikální úsečky s počátkem v každém vrcholu. Naivní přístup by položil přímku skrze vrchol v_i a poté se složitostí $O(n)$ prohledával všechny průsečíky

se všemi hranami polygonů e_j. Výsledná složitost $O(n^2)$ je příliš vysoká a bude tedy představena metoda *zametání přímkou*.

Zametání přímkou posouvá přímku zleva doprava a pozastavuje se na vrcholech. Těmto okamžikům říkáme *události*. Během posouvání si metoda uchovává list L, obsahující hrany, které přímka právě protíná. Změny tohoto listu se dějí pouze při událostech, ve kterých jsou zároveň určeny i vertikální úsečky z vrcholů.

Nalezení nejbližších průsečíků přímky s hranami z listu L by trvalo $O(n)$ jednoduchým prohledáním. Pokud bude ovšem list realizován efektivní strukturou, jako je například vyvážený strom, časová složitost se sníží na $O(\log n)$. Určit průsečík vertikální přímky s libovolnou hranou je jednoduché. Složitější je však najít průsečík takových hran e_{POD} a e_{NAD}, mezi kterými leží aktuální vrchol v, důvod události.

Definujme si ještě jednu dvojici hran: e_{pod} a e_{nad}, jako hrany obsahující vrchol v (různé od e_{POD} a e_{NAD}). Tyto dvě hrany se odlišují tak, že druhý vrchol hrany e_{pod} má y-ovou souřadnici nižší než druhý vrchol e_{nad}. Příklad všech zmíněných hran je na obrázku 5.5.



Obrázek 5.5: Příklad hran důležitých pro určení vertikálních úseček z aktuálního vrcholu.

Při zametání přímkou mohou nastat čtyři typy událostí, které vyžadují vykonání následujících akcí nad listem L (viz také obrázek 5.6):

1. e_{pod} a e_{nad} jsou obě nalevo od přímky
   (a) smazat e_{POD} a e_{NAD} z L

$$(\ldots, e_{POD}, e_{pod}, e_{nad}, e_{NAD}, \ldots) \rightarrow (\ldots, e_{POD}, e_{NAD}, \ldots)$$

2. e_{pod} a e_{nad} jsou obě napravo od přímky
   (a) vložit e_{pod} a e_{nad} do L

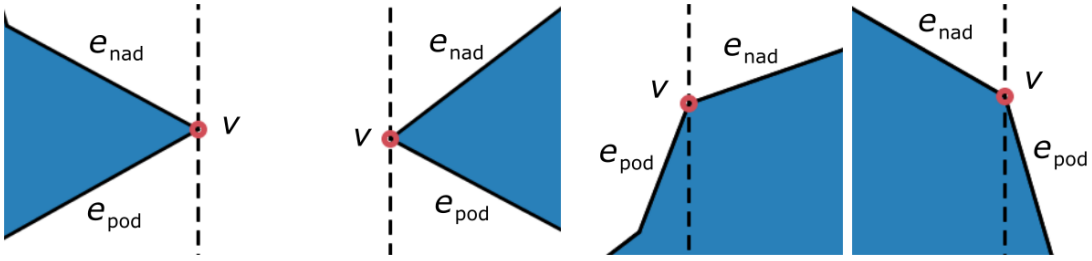
$$(\ldots, e_{POD}, e_{NAD}, \ldots) \rightarrow (\ldots, e_{POD}, e_{pod}, e_{nad}, e_{NAD}, \ldots)$$

3. e_{pod} je nalevo a e_{nad} je napravo od přímky
   (a) smazat e_{pod} z L a
   (b) vložit e_{nad} do L

$$(\ldots, e_{POD}, e_{pod}, e_{NAD}, \ldots) \rightarrow (\ldots, e_{POD}, e_{nad}, e_{NAD}, \ldots)$$

4. e_{pod} je napravo a e_{nad} je nalevo od přímky
   (a) smazat e_{nad} z L a
   (b) vložit e_{pod} do L

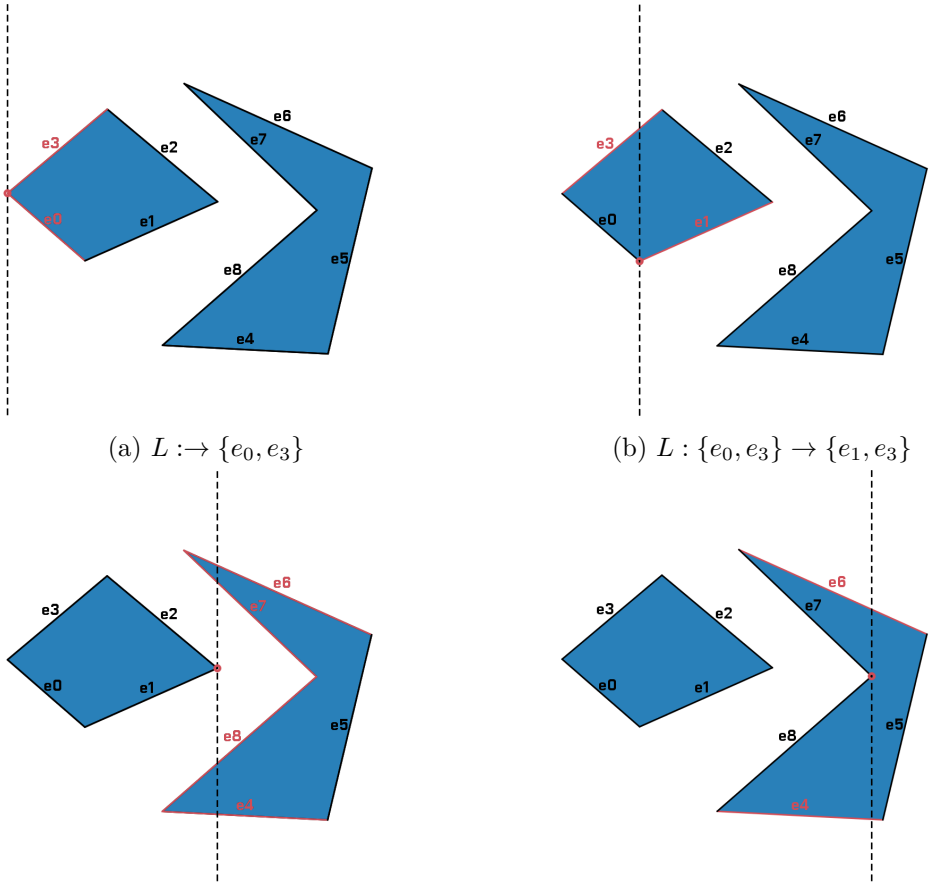
$$(\ldots, e_{POD}, e_{nad}, e_{NAD}, \ldots) \rightarrow (\ldots, e_{POD}, e_{pod}, e_{NAD}, \ldots)$$



(a) (b) (c) (d)

Obrázek 5.6: Typy událostí při metodě zametání.

S takto aktualizovaným listem L jsme schopni při každé události efektivně určit, s jakými hranami vypočítat průsečíky, abychom získali vertikální úsečky pro daný vrchol. Ukázku událostí a odpovídajících aktualizací listu je možné vidět na obrázku 5.7.



(a) $L :\rightarrow \{e_0, e_3\}$ (b) $L : \{e_0, e_3\} \rightarrow \{e_1, e_3\}$

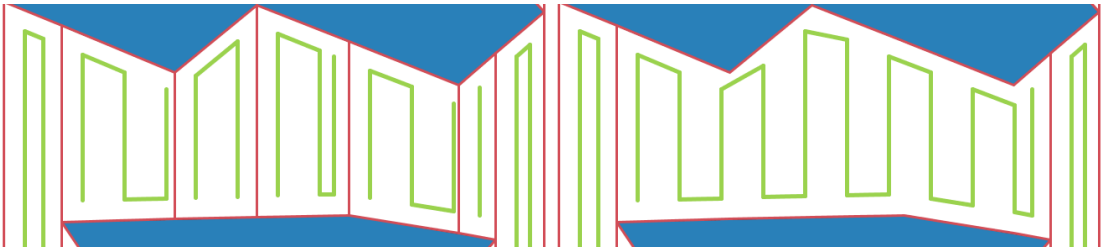
(c) $L : \{e_4, e_8, e_1, e_2, e_7, e_6\} \rightarrow \{e_4, e_8, e_7, e_6\}$ (d) $L : \{e_4, e_8, e_7, e_6\} \rightarrow \{e_4, e_6\}$

Obrázek 5.7: Příklady událostí a odpovídajících aktualizací listu L.

5.2 Boustrophedon dekompozice

Dosud popisované algoritmy pro hledání cesty určovaly trasu mezi počáteční a koncovou konfigurací. Podívejme se nyní na postupy, jejichž cílem je efektivně pokrýt celý dostupný prostor cestami tak, aby agent pohybující se po nich, svým efektorem o předem dané velikosti, zasáhl do každého přístupného místa. Pro jednoduchost uvažujme efektor kruhového tvaru.

Buněčné dekompozice jsou ideální pro tvorbu pokrývající cesty. Větší buňky jsou preferované, protože každé ukončení buňky, a tedy přerušení předem určeného, symetrického pohybu, znamená prodloužení výsledné cesty a více prostoru pro chyby při plánování. To je také důvod, proč takové cesty nebyly popisovány již u lichoběžníkové dekompozice. Lichoběžníkové buňky samotné se dají pokrýt efektivně, avšak s komplexnějším prostředím je jejich počet velký a větší robot by pokrýval opakovaně velkou část prostoru (viz obrázek 5.8). Tato neefektivita je určena z poměru pokryté plochy k délce celkové cesty.



(a) (b)

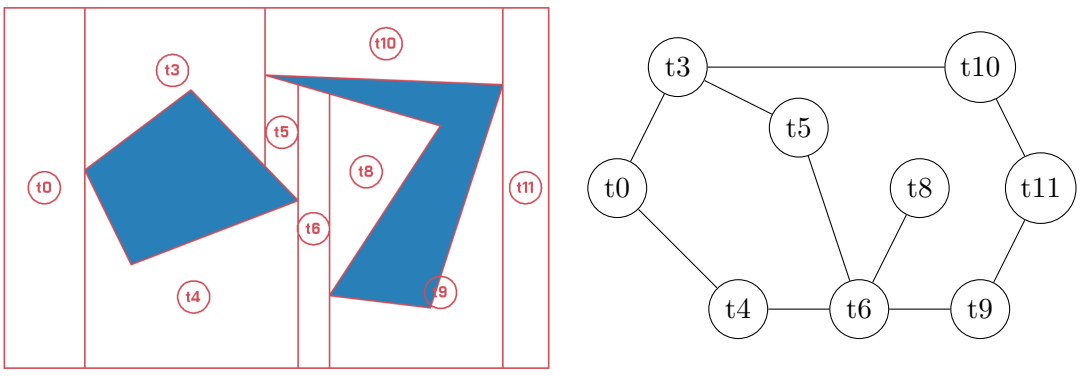
Obrázek 5.8: Pokrývání stejného prostoru za pomoci lichoběžníkových buněk (5.8a) a pokrývání jediné spojené buňky (5.8b).

Povšimněme si na obrázku 5.8b, že některé sousedící lichoběžníkové buňky mohou být jednoduše propojeny odstraněním společných hran. Získáme tak větší buňku, kde je možné se pohybovat s maximální vzdáleností od minulých pozic. Jedinou cenou je mírně zvýšená složitost buněk po stranách k překážkám. Pohyb kolem takto se měnící hrany překážky však pro robota se znalostí prostředí není problém.

Boustrophedon[2] dekompozice realizuje právě takovou transformaci lichoběžníkových buněk. Při zametání přímkou uvažuje pouze takové vrcholy, ve kterých je možné vytvořit vertikální úsečky v obou směrech y-ové osy. Nazvěme tyto vrcholy kritickými body dostupného prostoru.

Buňky jsou opět tvořeny prostorem mezi následnými úsečkami. S takto definovanou dekompozicí jsme poté schopni vytvořit graf sousednosti. Namísto hledání cesty v grafu z počáteční do cílové buňky, našemu záměru odpovídá průchod grafem generovaný například prohledáváním do hloubky, který obsahuje každou buňku alespoň jednou.

S jistotou pokrytí každé nalezené buňky zbývá určit cesty uvnitř jednotlivých buněk. Tyto cesty jsou tvořeny posloupností paralelních úseček ve vzdálenosti odpovídající šířce efektoru robota a krátkými spojnicemi mezi nimi, paralelní k hraně překážky (viz obrázek 5.8).



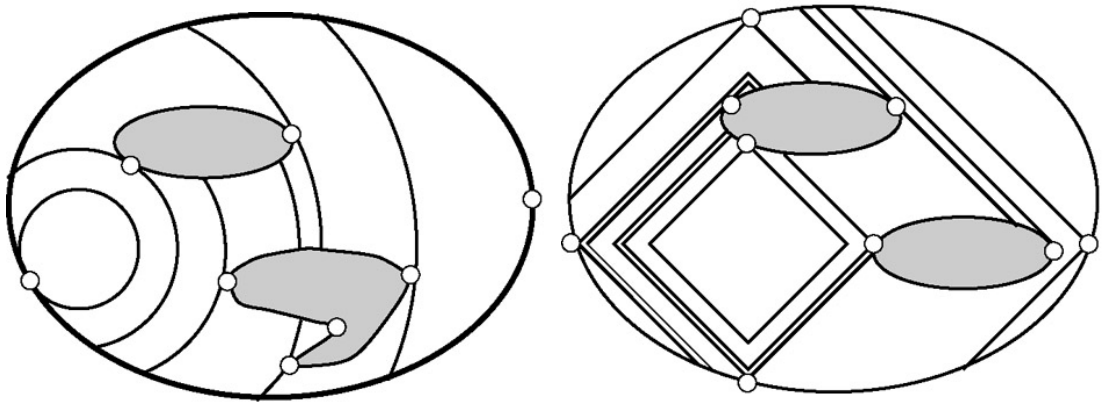
(a) (b)

Obrázek 5.9: Boustrophedon dekompozice dostupného prostoru (5.9a) a odpovídající graf sousednosti (5.9b).

[2]Boustrophedon (bustrofédon) je řecký výraz pro *obracení volů* popisující opakující se cestu jedním směrem následovanou cestou zpět o řádek dále využívaný například u stylu psaní.

5.3 Morseovy dekompozice

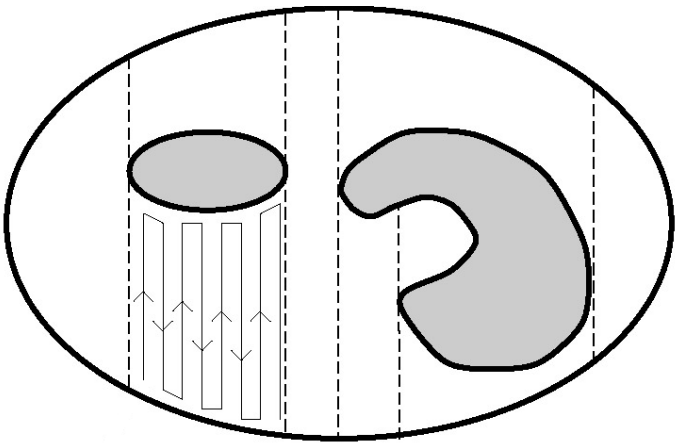
Boustrophedon dekompozici jsme schopni generalizovat mimo pouhé polygony díky technice rozřezání prostředí (slicing) představenou poprvé Cannym [1] v algoritmu pro tvorbu cestovních map (roadmap). *Řezem* (slice) je zde myšlen libovolný jedno-dimenzionální útvar, který je možné parametrizovat tak, aby se změnou parametru postupně procházel (zametal) celým prostorem. Takovým řezem je například přímka definovaná v předchozích částech této kapitoly. Může se však také jednat o libovolný n-úhelník, kružnici nebo pouze její část (viz obrázek 5.10).



(a) (b)

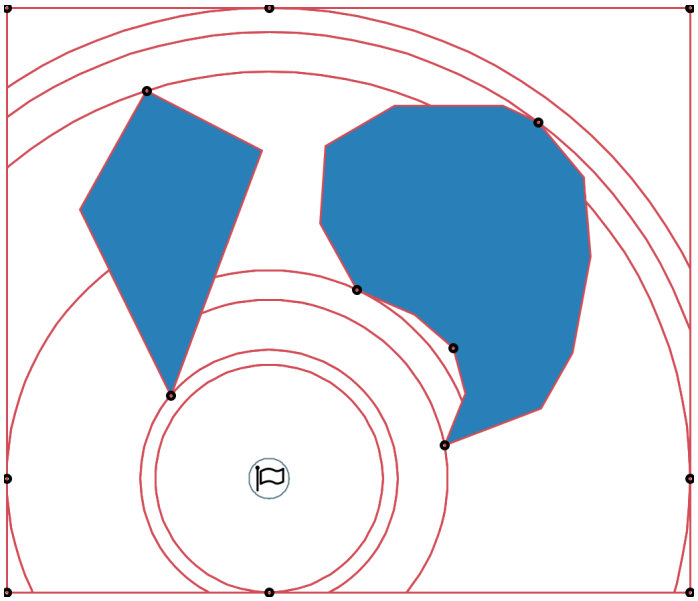
Obrázek 5.10: Dekompozice vytvořená kruhovým (5.10a) a čtvercovým (5.10b) řezem. [2]

Jak je řez posouván skrze dostupný prostor, naráží na překážky a je jimi dělen na menší části, které se později opět spojují. Tyto body, kde se mění spojitost řezu jsou právě kritickými body, které jsme si definovali u Boustrophedon dekompozice. Vzhledem k odlišnému tvaru řezu se však nemusejí vyskytovat pouze ve vrcholech překážek, ale v libovolném bodě na jejich vnější hranici (viz obrázek 5.11). Tato vlastnost není nežádoucí vzhledem k tomu, že se snažíme upustit od prostředí definovaného polygony (tedy množinou vrcholů).



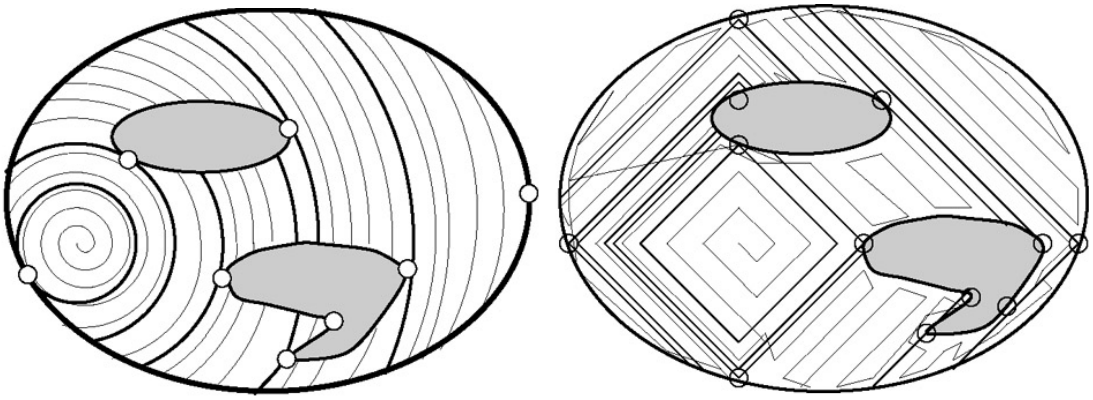
Obrázek 5.11: Boustrophedon dekompozice nepolygonálního prostředí a pokrývající vnitřní cesta buňky. [2]

Vzhledem k tomu, že se snažíme pokrýt okolí i nekonvexních překážek, je třeba definovat další kategorii kritických bodů. Tyto body jsou v nejvzdálenějším místě dostupných konkávních *zátok* překážek. Z hlediska procházejících řezů se nachází v místě, kde určitá spojitá část řezu kompletně zaniká (viz pravá překážka na obrázku 5.12).



Obrázek 5.12: Řezy v kritických bodech vytvářející kruhovou dekompozici.

Řezy v kritických bodech nazvěme kritickými řezy. Množina těchto řezů je u Boustrophedon dekompozice rovna množině všech vertikálních úseček. Stejně jako jsme u tohoto rozložení definovali buňky následnými úsečkami, jsme schopni najít libovolnou buňku Morseovy dekompozice mezi následovnými spojitými částmi kritických řezů. Můžeme dokázat, že uvnitř každé takto vytvořené buňky nemůže dojít k rozdělení nebo spojení řezu a jeho topologie tedy zůstává konstantní. Robot je tedy opět schopen pokrýt buňku střídáním pohybu po řezu s pohybem kolem překážky, dokud není znovu ve vzdálenosti jedné šířky od minulého pohybu po řezu. Z těchto poznatků samozřejmě vyplývá, že i Boustrophedon dekompozice patří mezi Morseovy dekompozice.



(a) (b)

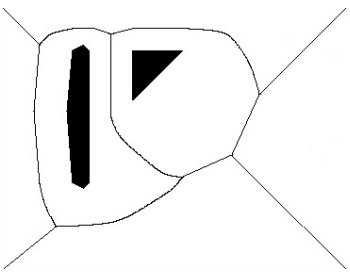
Obrázek 5.13: Spirálová (5.13a) a čtvercová (5.13b) cesta, kterou je možné použít k aproximaci spirály. [2]

Pokud je naším cílem vytvořit cestu maximalizující poměr pokryté plochy vzhledem k její vzdálenosti, je vhodné, namísto střídání pohybu po (například) kruhovém řezu a pohybu vně, se rovnou pohybovat po spirále. Na obrázku 5.13 je možné vidět příklady takového přístupu, který je efektivnější v prostředí s malým počtem překážek. Další výhodou tohoto přístupu k vytváření výsledné cesty je fakt, že robot, který je schopen se takto konstantně pohybovat, může prostředí pokrývat a objevovat ve stejnou chvíli. Můžeme tedy vytvářet dekompozici neznámého prostředí za pomocí senzorů.

5.4 Brushfire dekompozice

Algoritmus Brushfire jsme definovali v kapitole 4 na mřížce, kde nám umožňoval vypočítat vzdálenost v počtu buněk od nejbližší překážky. Vzhledem k tomu, že se snažíme vytvářet *exaktní* dekompozice dostupného prostoru, popišme jej nyní na spojité doméně.

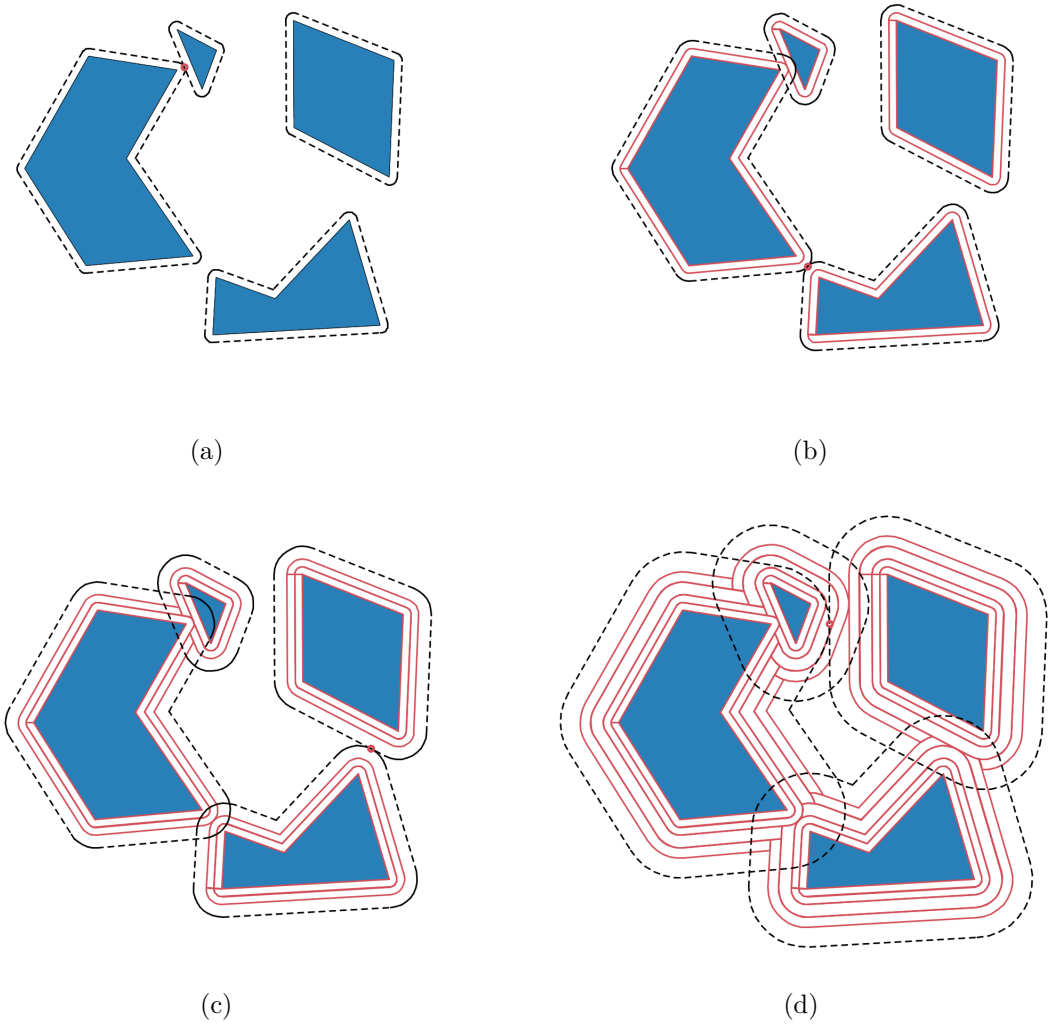
Cílem zůstává najít pro každý bod vzdálenost od nejbližší překážky, tedy spojitou funkci vzdálenosti, kde body se stejnou hodnotou tvoří vlnu podobnou lesnímu požáru (nebo žďáření) s počátkem na hranici překážky. Linie, na kterých se tyto vlny setkávají, odpovídají dekompozici GVD[3] (viz obrázek 5.14). Tato dekompozice je odlišná od těch, které zde hledáme v tom, že nijak nepomáhá při tvorbě pokrývajících cest. Je však důležitou součástí cestovních map a stojí tedy za zmínku. Místo toho definujme odlišnou dekompozici, která žádané cesty vytvořit umožní.



Obrázek 5.14: Voroného dekompozice prostředí s překážkami. [2]

[3]Generalized Voronoi Diagrams (Voroného diagram)

Funkce vzdálenosti D, která měří vzdálenost mezi bodem v prostředí a nejbližším bodem nejbližší překážky vytváří Brushfire dekompozici. Každý řez funkce D je vlna, kde každý bod na její hranici je právě λ daleko od nejbližší překážky. Se zvětšující se λ se řez posouvá dále prostředím. Uvažujeme-li takovéto propagující se vlny od každé překážky, které jsou v prostředí alespoň dvě, nastane situace, kdy do sebe dvě vlny narazí. V okamžiku každého takového nárazu je vytvořena množina nových buněk definovaných maximální vzdáleností λ od překážek, ze kterých jejich propagace započala (viz obrázek 5.15). Pro jednoduchost uvažujme, že prostředí je konečné a tak i v případě osamocené překážky jsme schopni eventuálně propagaci ukončit a buňku uzavřít.

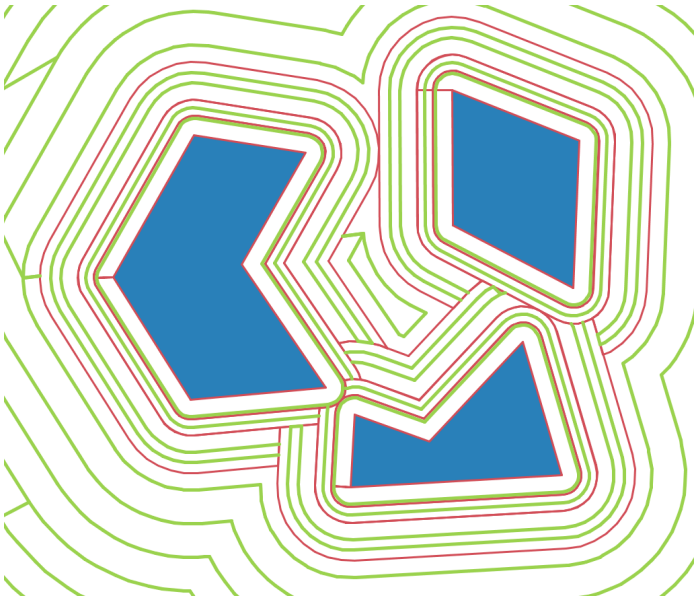


Obrázek 5.15: Příklad postupného vytváření buněk dekompozice Brushfire.

Podle nalezených buněk vytvoříme graf sousednosti a následně určíme průchod grafem stejně jako v minulých případech. Cesta pokrývající buňku tentokrát sestává z pohybu po řezu (vlně) v dané vzdálenosti od počáteční překážky, pohybem kolmým na řez, dokud nejsme o šířku efektoru robota dále, a nakonec pohybem kolem hranice buňky.

Robot se pohybuje po řezu sledováním cesty ve stále stejné vzdálenosti od překážky, dokud nedokončí celý okruh nebo nenarazí na bod, kde vzdálenost ke dvěma překážkám je

stejná. Pokud narazí na svůj počáteční bod okruhu, posune se o jednu šířku dále od překážky a opakuje pohyb kolem překážky v nové vzdálenosti. Ve chvíli kdy narazí na hranici buňky, sleduje ji, dokud se nedostane do vzdálenosti od překážky odpovídající násobku jeho šířky. V tu chvíli pokračuje v pohybu kolem překážky v této vzdálenosti.



Obrázek 5.16: Pokrytí buněk Brushfire dekompozice.

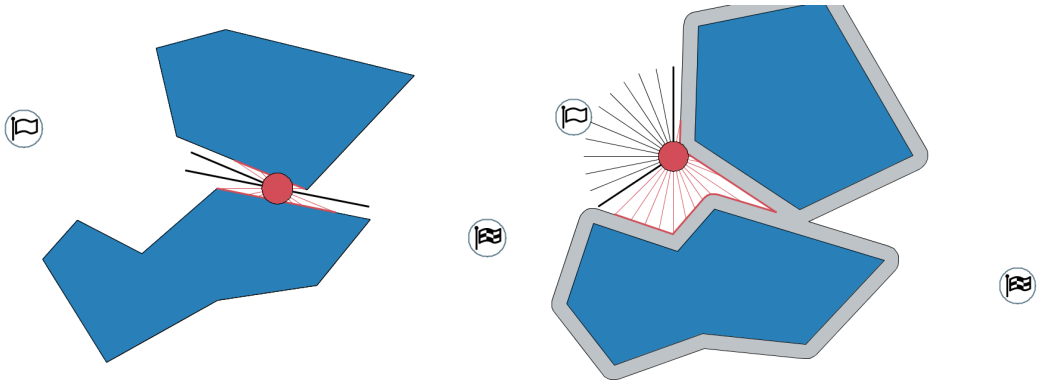
Kapitola 6

Simulace robota se senzorem vzdálenosti

Bug a jiné na senzorech závislé plánovací algoritmy předpokládají, že ovládají robota schopného pohybu kupředu a pohybu po obvodu překážky v dané vzdálenosti. První z těchto pohybů je triviální, ať už se jedná o krok směrem k cíli nebo po tečně překážky. Stačí vědět, že, alespoň ve vzdálenosti jednoho kroku, nedojde ke kolizi s překážkou.

Druhý případ už je výzvou, protože hrana překážky není předem známá. Robot musí využít informací ze senzorů a to nejen z jednoho směru. K odvození správného lokálního pohybu je potřeba řada informací: velikost robota, délka jeho diskrétního kroku, dosah a rozlišení senzoru, vzdálenost robota od nejbližšího bodu obcházené překážky a směr a odvozená normála překážky v tomto bodě. V následující části textu bude popsán způsob nalezení dalšího kroku kolem překážky, tak jak byl implementován v demonstrační aplikaci, a vliv těchto proměnných na něj.

6.1 Velikost robota

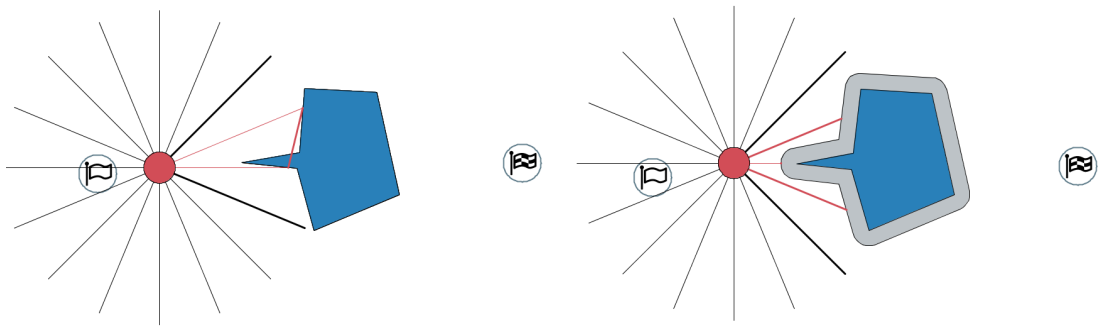


(a) (b)

Obrázek 6.1: Příklad situace, kdy velký robot není schopen projít úzkým průchodem mezi dvěma překážkami (6.1a) a podobná situace, kdy známý dostupný konfigurační prostor umožňuje tento fakt detekovat (6.1b).

Senzorem řízené algoritmy předpokládají robota o velikosti bodu. Díky tomu mohou zaručit, že příkaz jako:„postupuj kolem hrany překážky“ bude vykonán bez sebemenšího problému, protože mezi dvěma překážkami je vždy dostatek místa pro jednoduchý bod. Jinak by se překážky překrývaly a byly by tedy spojeny do jedné.

U robota libovolné velikosti může dojít k situacím, kdy mezi dvěma překážkami není dostatek místa a robot není schopen před tím, než krok udělá, s jistotou předpovědět, že nenarazí do jedné z nich. V případě úzkého průchodu mezi dvěma překážkami (viz obrázek 6.1) je důvodem snaha o jednoduchost a pochopitelnost implementace, avšak existují i jiné, vážnější situace. Na obrázku 6.2 je přiblížen problém, kdy rozlišení senzoru je výrazně nižší, než rozlišení překážky, která obsahuje jemné prvky způsobující kolizi.



(a) (b)

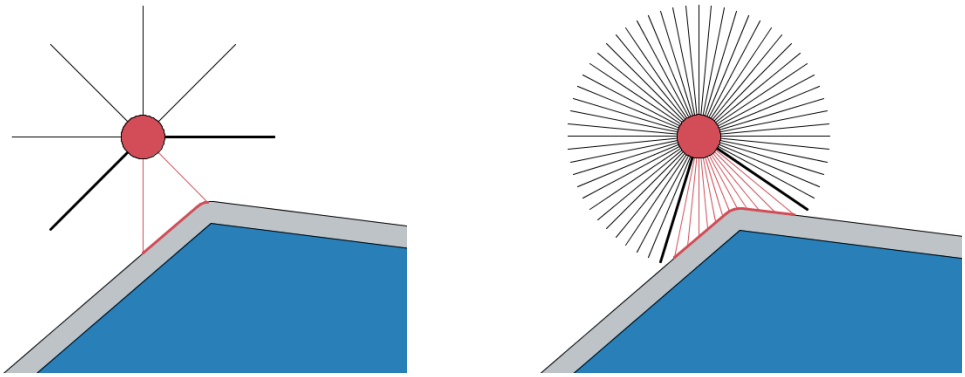
Obrázek 6.2: Překážka s jemným výstupem, který senzor nezaznamená vzhledem k jeho diskretizaci (6.2a), body na překážce jsou spojeny přímo, tak jak je vidí robot. Podobná situace, kde detekovanou překážkou je dostupný konfigurační prostor (6.2b).

V obou případech je druhým obrázkem (6.1b a 6.2b) naznačeno řešení, které bylo implementováno v této práci. Jedná se o záměnu dostupného konfiguračního prostoru s celým konfiguračním prostorem, která efektivně rozšiřuje každou překážku právě o polovinu velikosti robota. Toto řešení je sice v rozporu s ideou, že senzorem řízené algoritmy pracují v neznámém prostředí, avšak cílem této práce je jejich prezentace. Simulace jiného než bodového robota není v tomto rozsahu možná a popsaná transformace prostředí umožňuje, alespoň v rámci možností, realizovat robota libovolné velikosti.

6.2 Senzor

Pro algoritmy třídy Bug jsou typické senzory, které využívají podobně jako slepý brouk (bug) tykadla, aby jejich pomocí hmatem objevovaly překážky a vyhýbaly se jim. Tyto senzory mohou mít dosah od nuly, kdy se jim říká dotykové, až do nekonečna, které vidí na rozdíl od brouků do jakékoli vzdálenosti. Senzory s nekonečným dosahem bývají spárovány s omezeným prostředím simulujícím horizont dostupného prostoru nebo nahrazeny dosahem rozumně velkého čísla. Kromě dosahu mají senzory ještě jednu důležitou vlastnost – rozlišení. Ve výpočetním světě je potřeba mnohé atributy diskretizovat a rozlišení takové diskretizace může výrazně ovlivnit výsledek. Jak nespojitost kroků robota tak rozlišení senzoru ovlivňuje samotnou délku nalezené cesty. Pokud se totiž robot pohybuje podél pře-

kážky, jejíž směr se neslučuje precizně se směrem a tvarem paprsku senzoru, musí neustále korigovat směr pohybu, aby se udržel ve správné vzdálenosti od překážky.



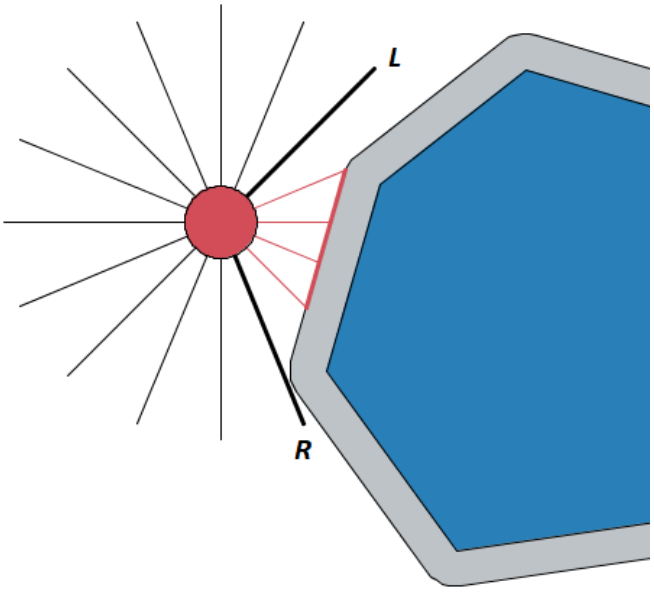
(a) (b)

Obrázek 6.3: Sensor s rozlišením 8 paprsků (6.3a) a 32 paprsků (6.3b)

6.3 Pohyb kolem překážky

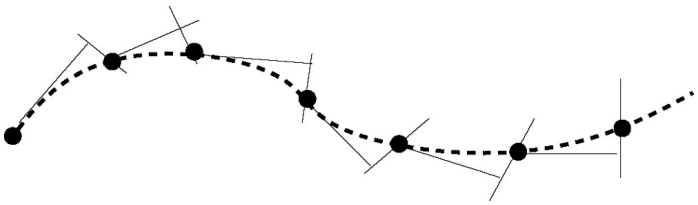
Postupy dosud popsané v této kapitole, nám umožňují robota zjednodušit na bod pohybující se kolem hrany překážky. Nyní se podívejme, v jakém směru by se robot měl v dalším kroku vydat, aby se udržoval, se svými diskrétními kroky, v co možná nejmenší vzdálenosti od překážek. Pokud bychom uvažovali robota libovolné velikosti, pak nalezení normály k nejbližšímu bodu překážky je popisovaný ([2], kapitola 2) postup k určení směru pohybu (viz obrázek 6.4). Robot pak neustále nalézá lokální tečny k překážkám. V sekci 6.1 však byly představeny problémy spojené s tímto přístupem.

Bodový robot umožňuje mnohem robustnější styl pohybu. Namísto tečny v nejbližším bodě uvažujme tečnu k překážce, kterou vidí sám robot, podobně jako u Tangent Bugu popisovném v kapitole 3. Stačí nalézt první nespojitost ve zpětné vazbě senzoru směrem doleva, resp. doprava (dle směru obcházení překážky), a víme, že alespoň v bodě této tečny bude robot přímo na hranici překážky. Pokud pak na rozdíl od Tangent Bugu započneme tento styl pohybu až v přímé blízkosti překážky se senzorem, který vidí pouze do vzdálenosti odpovídající robotově velikosti, pohybuje se virtuálně po hranici překážky.



Obrázek 6.4: Směr pohybu odvozený od první levé (L), resp. pravé (R), nespojitosti.

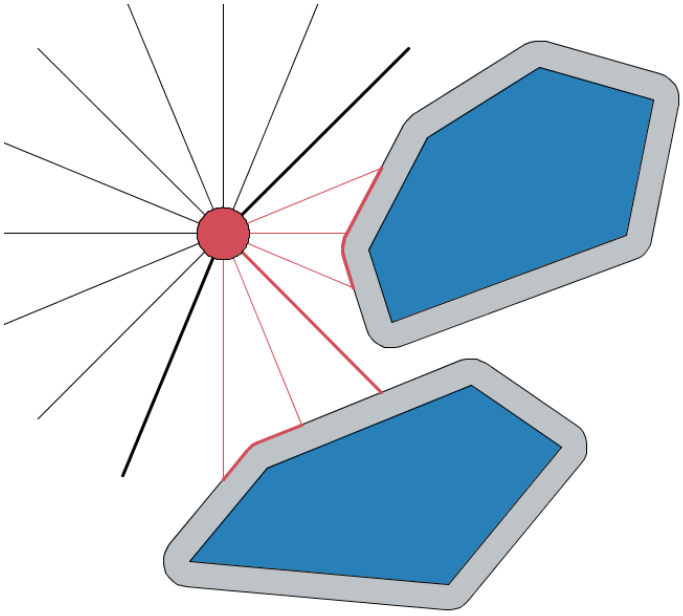
Simulovaný pohyb je pouze aproximací popsané metody, jejíž přesnost závisí na jemnosti diskretizace kroku. Ve skutečnosti robot vytváří cestu, která vždy opustí hranici překážky a následně se k ní korekcí vrátí (viz obrázek 6.5). Výhodou popisované metody využívající aktuálních tečen je v tom, že tyto korekce jsou automatické a není potřeba v žádné případu kontrolovat, zda se robot posledním krokem nedostal do prostoru překážky.



Obrázek 6.5: Aproximace reálné cesty způsobená diskretizací kroků. Každý krok je schopen odklonit robota od hranice překážky, což vyžaduje následnou korekci. [2]

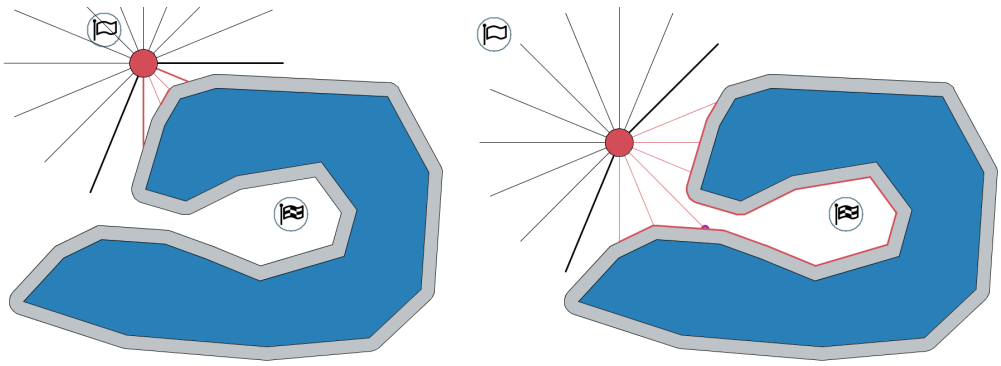
6.4 Problém nalezení nespojitostí ze senzoru

Nespojitosti ve zpětné vazbě senzoru jsou hledány podle vzdálenosti, kterou daný paprsek senzoru urazil, než narazil na hranici nějaké překážky. Rozdíly v hodnotách mezi následnými paprsky udávají, jestli se jedná o spojitý nebo nespojitý přechod. Samozřejmými případy nespojitosti je rozdíl mezi paprskem, který nenarazil na žádnou překážku a paprskem, který narazil. Dochází však k případům, kdy jedna překážka překrývá část druhou a tak poslední paprsek na první naráží do překážky stejně jako první paprsek na druhé (viz obrázek 6.6).



Obrázek 6.6: Případ nespojitého skoku (tlustá, červená) mezi dvěma paprsky dopadající na překrývající se překážky.

Zde je potřeba určit prahovou vzdálenost, jejíž překročení mezi následnými paprsky je označí za nespojité. Byly experimentálně testovány mnohé přístupy, mezi které patřily mimo jiné konstantní hodnota nebo proměnná v závislosti na velikosti robota a jeho kroku. V určitých situacích však všechny přístupy vyvolaly buď false pozitivy (viz 6.7a) nebo false negativy (6.7b).



(a) (b)

Obrázek 6.7: False positivy (6.7a) a false negativy (6.7b) způsobené nevhodnou volbou prahu nespojitostí.

Hodnota tohoto prahu byla tedy označena za vstupní parametr *continuity jump* algoritmů, pro které je relevantní, s počáteční hodnotou odpovídající velikosti robota. Velikosti, jejíž nespojitý skok zaručuje průchod v překážce odpovídající minimálně velikosti robota.

Kapitola 7

Demonstrační aplikace

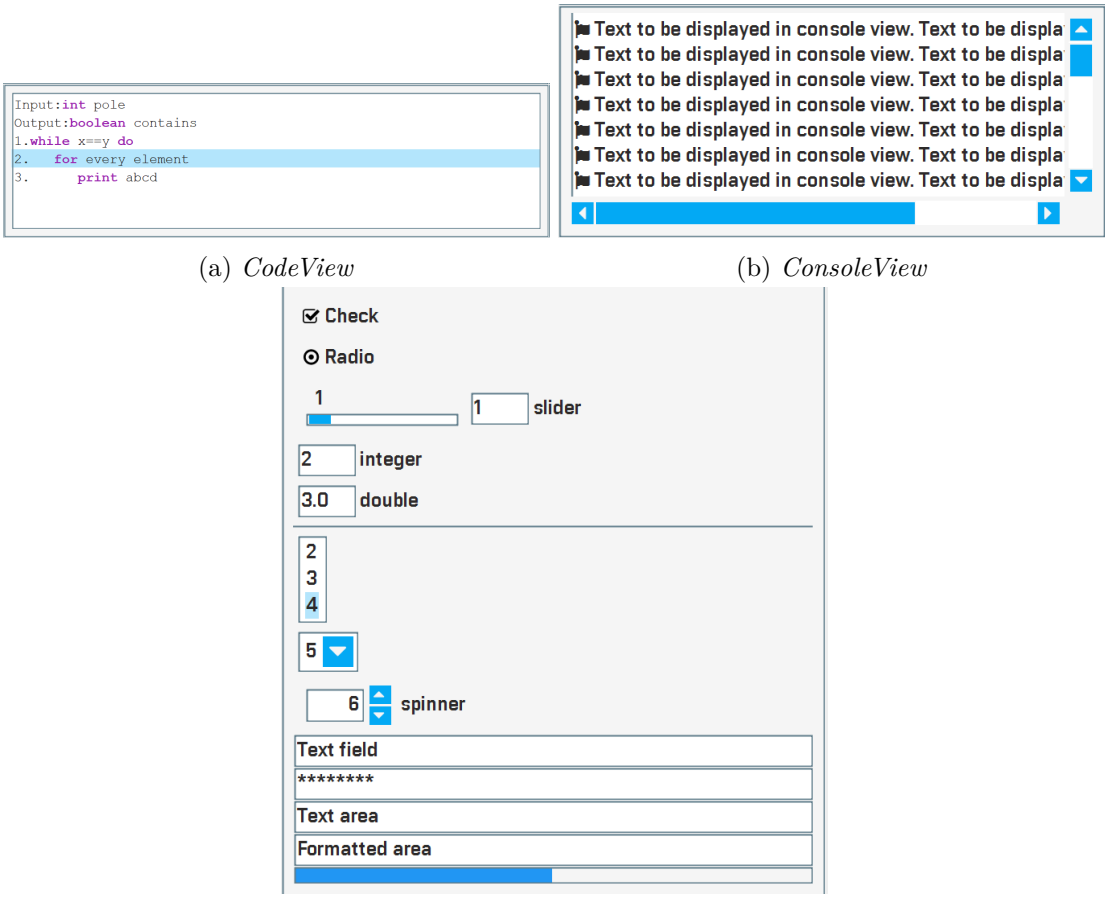
V této kapitole bude popsána samotná aplikace demonstrující funkci algoritmů, které byly popsány v předchozích kapitolách.

Na rozdíl od aplikací vytvořených Jakubem Rusnákem [5], tento program zahrnuje všechny algoritmy současně. Důvodem pro tuto změnu je schopnost takové aplikace porovnávat jednotlivé běhy a dokonce celé metody proti sobě na stejných mapách. Nebude potřeba ukládat mapy a řešení *offline* ve zjednodušených reprezentacích, pokud to nebude vyžadováno, a tím bude menší šance ztráty důležitých informací.

Algoritmy plánování cesty byly realizovány tak, aby jak vizuálně tak svojí implementací v Javě odpovídaly teoretickému popisu z předešlých kapitol. Tím mělo být dosaženo bezproblémového přechodu z teorie do praxe a jednoduchého experimentování s algoritmy, které jsou popsány v učebnicích.

7.1 Knihovna pro vizualizaci

Pro implementaci navrhovaných řešení plánování cesty byla vybrána knihovna *vizlib*, kterou vytvořil Jakub Rusnák [5] v jazyce Java pro demonstrační účely. Knihovna abstrahuje funkční celky uživatelského rozhraní – zobrazení, které umožňují rychle ukazovat postup algoritmu na základě vizualizace konfiguračního prostoru (*MapView*), krokování textového popisu (*CodeView*), nadefinovaných textových výstupů (*ConsoleView*) a ovládání simulace (*ToolbarView*).

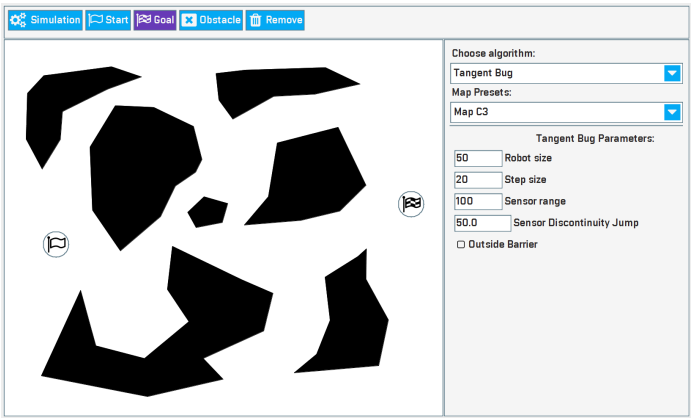


(a) *CodeView* (b) *ConsoleView*

(c) *ParameterView*

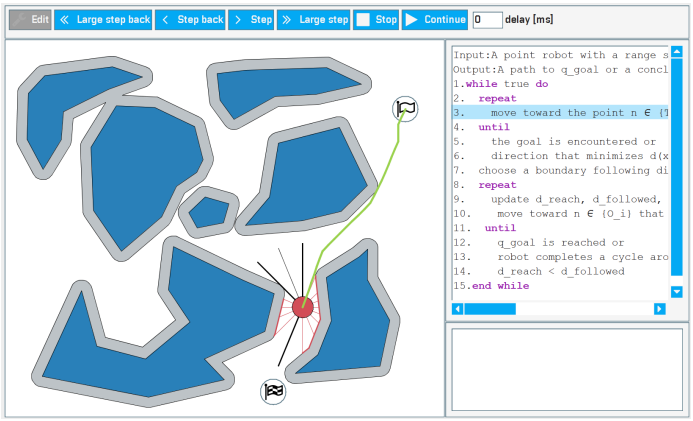
Obrázek 7.1: Ukázka zobrazení implementovaných v knihovně *vizlib*. [5]

Aplikace běží ve dvou režimech. V editovacím režimu je možné vytvořit libovolnou mapu s překážkami (případně zvolit jednu z testovacích map) a zadat parametry algoritmu v *ParameterView* (viz 7.2).



Obrázek 7.2: Ukázka aplikace v režimu editace.

Druhým režimem je samotný běh algoritmu, který nabízí vizualizaci běhu. Dále je po pravé straně zobrazení pseudokódu a konzole vypisující informace o průběhu algoritmu. Hlavním důvodem použití této knihovny je implementace krokování algoritmů, které je zde nejdůležitějším prvkem.



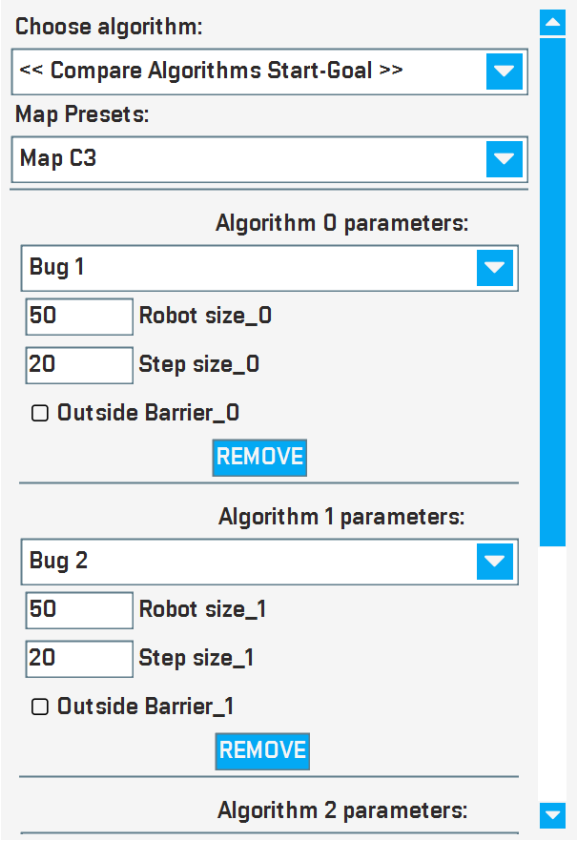
Obrázek 7.3: Ukázka aplikace v režimu běhu.

Práce Jakuba Rusnáka obsahuje také implementaci algoritmů plánování cest s odlišnými přístupy, které se zprvu zdály jako vhodné příklady. Vzhledem k tomu, že jsou decentralizované a špatně porovnatelné, příkladem bylo pouze použití řízení běhu.

7.2 Dávkové spouštění

Na listu algoritmů jsou dále možnosti *Compare Algorithms Start-Goal* a *Compare Algorithms Coverage*, které v *ParameterView* umožňují volbu libovolného množství běhů a jejich parametrů. Tlačítkem *Add* je přidán běh, ve kterém je možné zvolit algoritmus z dané kategorie a jeho parametry. *Remove* pak daný běh odstraní.

Simulace probíhá tak, že v případě porovnání navigačních algoritmů jsou výsledné cesty postupně zobrazovány pro všechny vybrané běhy. Tyto trasy jsou označeny náhodnou barvou a jménem *aN*, kde *N* označuje index daného běhu. V případě porovnání algoritmů pokrytí jsou trasy zobrazovány postupně, aby nedocházelo k přílišnému křížení cest. Výsledné statistiky běhů jsou vypisovány do *ConsoleView*.

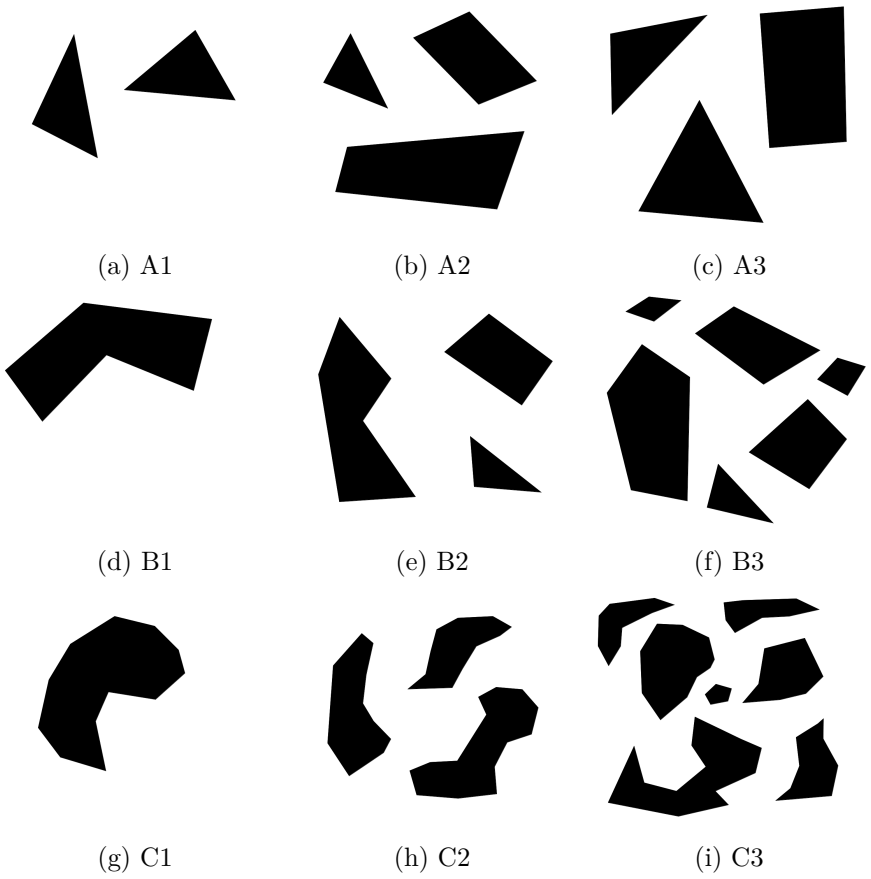


Obrázek 7.4: Volba parametrů při dávkovém spouštění

Kapitola 8

Srovnávací studie plánovacích algoritmů

Srovnání implementovaných algoritmů, popisované v této kapitole, proběhlo na sadě ručně vytvořených map. Jedná se o celkově 9 map, které obsahují překážky s postupně se zvyšující složitostí.



(a) A1 (b) A2 (c) A3

(d) B1 (e) B2 (f) B3

(g) C1 (h) C2 (i) C3

Obrázek 8.1: Mapy, na kterých proběhlo testování s proměnnou komplexností (A-C) a plochou překážek (1-3).

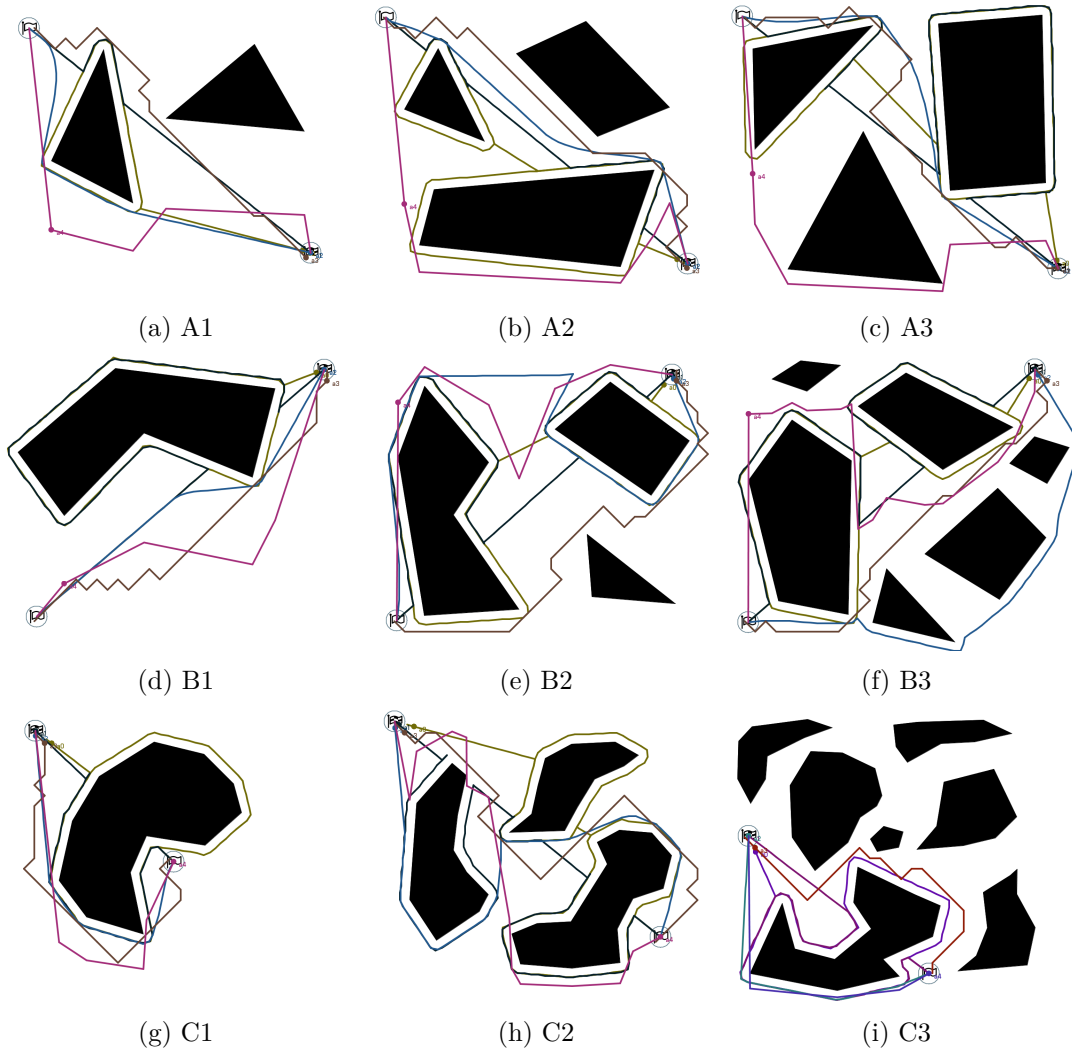
Vytvořeny byly na základě dvou řídících parametrů: komplexnost (A-C) a plocha (1-3). Mapy jsou tedy označeny podle parametrů A1 až C3 (viz obrázek 8.1). První z těchto parametrů ovlivňuje počet vrcholů a různorodost povrchu každé překážky. Plocha pak ovlivňuje procentuální pokrytí celého prostoru překážkami ve třídách: *10-15%*, *15-25%*, *25-35%*.

8.1 Navigace

Porovnání výstupů je možné pouze v rámci tříd algoritmů se stejným účelem. První kategorii tvoří algoritmy realizující navigaci:

- Bug1
- Bug2
- Tangent Bug
- Potenciální pole
- Lichoběžníková dekompozice

V rámci navigace je nejdůležitějším ukazatelem výkonu algoritmu délka nalezené cesty. Na obrázku 8.2 jsou zobrazeny cesty generované jednotlivými algoritmy na všech mapách. Tabulka 8.2 pak kvantifikuje cesty z hlediska jejich délky.



(a) A1 (b) A2 (c) A3

(d) B1 (e) B2 (f) B3

(g) C1 (h) C2 (i) C3

Obrázek 8.2: Mapy, na kterých proběhlo testování s proměnnou komplexností (A-C) a plochou překážek (1-3).

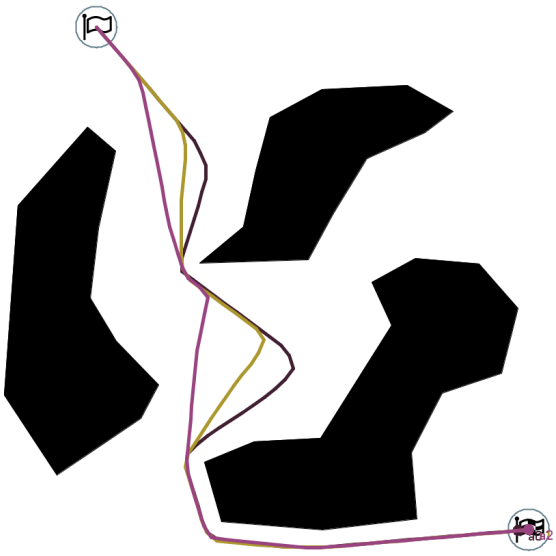
Mapa	Bug 1	Bug 2	Tangent Bug	Potenciálové pole	Lichoběžníková d.
A1	2557.55	1209.84	1243.43	1205.51	1485.86
A2	4214.65	1356.85	1208.9	1397.93	1764.17
A3	5406.39	3032.2	1342.32	1572.79	1737.37
B1	4619.76	2610.72	1147.5	1307.93	1295.97
B2	4876.72	2361.6	2173.63	1553.08	1825.95
B3	4708.66	2336.04	1619.29	1403.08	2049.74
C1	3190.47	1282.55	1055.1	1245.8	1196.08
C2	4919.62	3005.76	1939.34	1672.2	1762.45
C3	3038.67	1704.12	970.68	1083.38	968.25
Σ	37532.49	18899.68	12700.19	12441.7	14085.84

Tabulka 8.1: Výsledné délky cest navigačních algoritmů na jednotlivých mapách.

Napříč všemi mapami je zřejmé seřazení dle výkonnosti od nejhoršího po nejlepší: Bug 1, Bug 2, Lichoběžníková dekompozice, Tangent Bug, Potenciálové pole. Rozdíly jsou markantní s výjimkou dvou nejlepších algoritmů, které se v prvenství střídají na jednotlivých mapách. Odděluje je až suma všech běhů, stavící na první pozici Potenciálové pole.

8.2 Vliv dosahu senzoru na Tangent Bug

Dosud byly algoritmy testovány se základními hodnotami parametrů. Pro Tangent Bug to znamená, že dosah jeho senzoru byl limitován na 100 jednotek. Lze dokázat, že s dosahem přibližujícím se nekonečnu Tangent Bug nalézá optimální cestu. Na obrázku 8.3 je možné vidět postupné zkracování cesty pro dosah senzoru 50, 100 a 200.



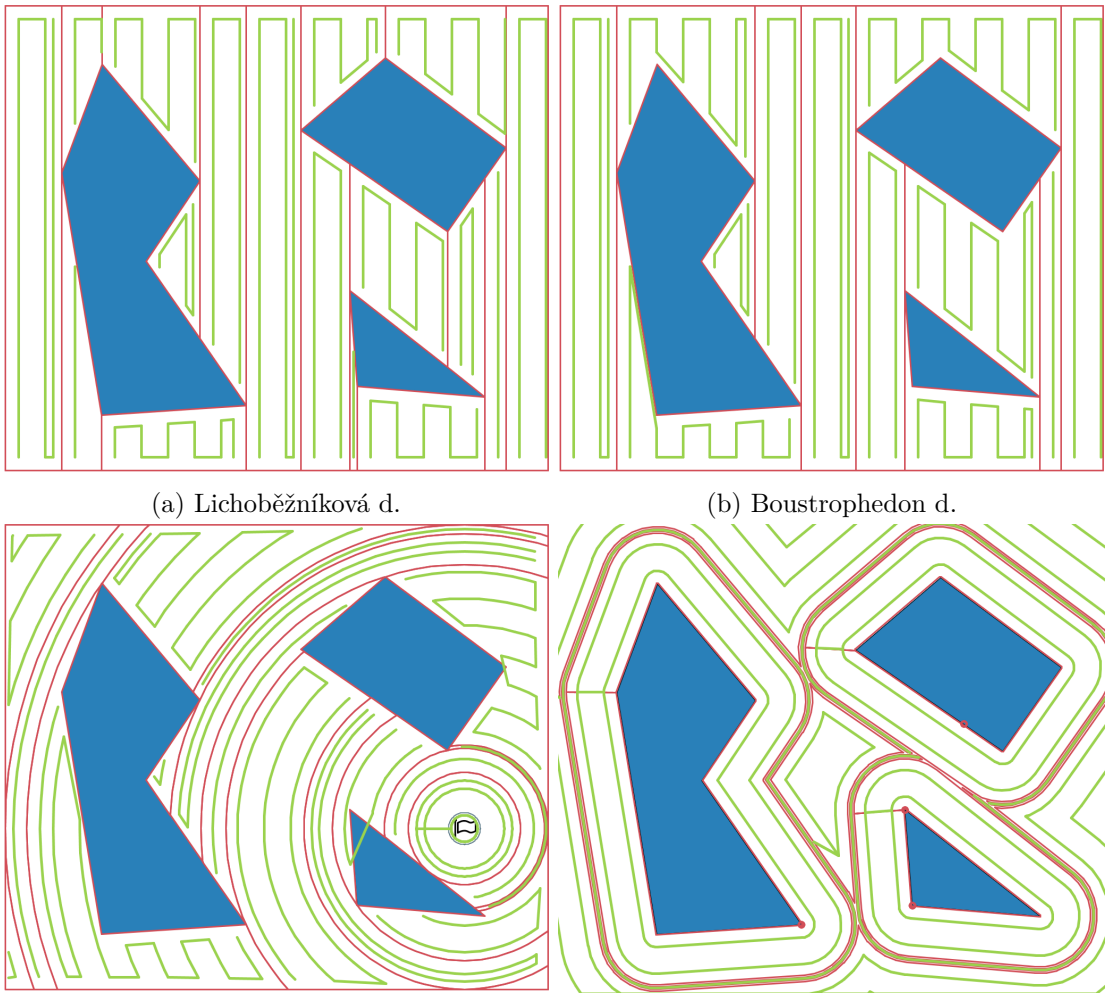
Obrázek 8.3: Ukázka vlivu dosahu senzoru na délku cesty generovanou Tangent Bugem.

Dosah senzoru	Délka cesty	Barva
50	1455.33	fialová
100	1377.62	žlutá
200	1266.70	hnědá

Tabulka 8.2: Data k cestám z obrázku 8.3.

8.3 Pokrytí

Druhou třídou podle užití jsou algoritmy realizující pokrytí. Stejně jako u navigace byla testována délka cesty potřebná pro pokrytí dostupného konfiguračního prostoru. Je zřejmé, že se zvyšující se délkou cesty dochází k překrývání cest efektoru a tedy zhoršujícím se řešením.



(a) Lichoběžníková d.

(b) Boustrophedon d.

(c) Kruhová d.

(d) Brushfire d.

Obrázek 8.4: Případy běhů algoritmů realizující pokrytí na mapě B2.

Mapa	Lichoběžníková d.	Boustrophedon d.	Kruhová d.	Brushfire d.
A1	17163.05	17340.63	24019.24	23335.13
A2	14150.92	14287.41	17304.4	29296.44
A3	14030.51	13399.59	26456.73	19574.6
B1	16053.77	15095.95	19059.15	18766.13
B2	16452.36	16847.34	18221.25	21321.69
B3	15799.35	14572.26	16439.6	41553.28
C1	17021.35	15947.44	19223.89	20661.88
C2	18073.28	16149.52	17751.74	26678.03
C3	18335.38	14520.8	23951.12	52980.32
Σ	147079.97	138160.94	182427.12	254167.5

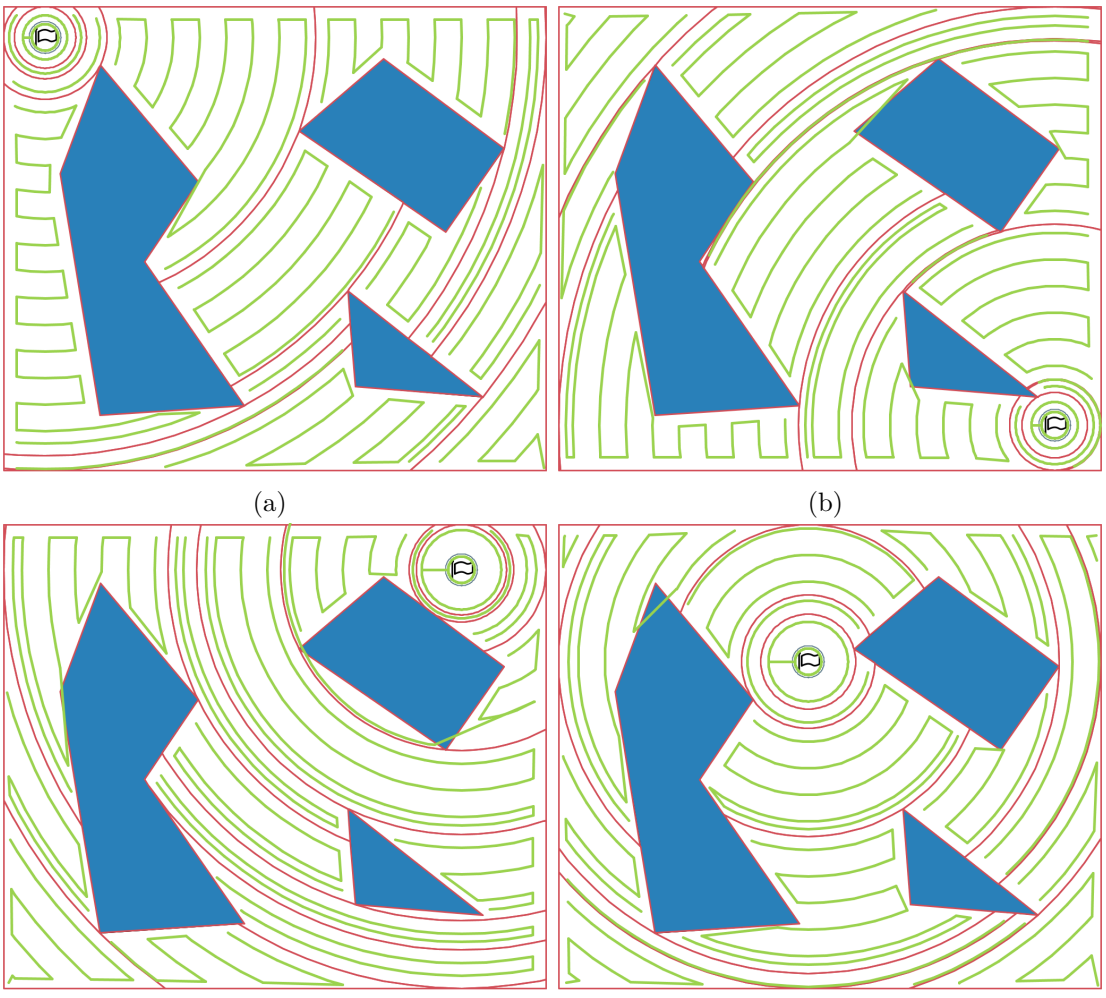
Tabulka 8.3: Výsledné délky cest algoritmů realizující pokrytí na jednotlivých mapách.

Ze získaných dat vyplývá, že Boustrophedon dekompozice je ideální pro pokrývání dostupného prostoru. Na dalších pozicích je lichoběžníková dekompozice, která je v každém

ohledu, až na implementaci, méně výkonná. Kruhová a Brushfire dekompozice generuje poměrně dlouhé trasy, avšak díky rozdílným řezům generují typy cest, které mohou být vhodné v závislosti na aplikaci.

8.4 Vliv počáteční konfigurace na Kruhovou dekompozici

Kruhová dekompozice se odlišuje od ostatních testovaných algoritmů možností definovat počáteční konfiguraci. Proto byla na mapě B2 zvolena množina startovních pozic, ze kterých byly vygenerovány následující cesty (viz obrázek 8.5)



Obrázek 8.5: Ukázka vlivu počáteční konfigurace na výslednou cestu generovanou kruhovou dekompozicí.

Pozice počátku	Délka cesty
8.5a	15167.34
8.5b	21249.35
8.5c	22138.06
8.5d	19726.80

Tabulka 8.4: Data k cestám z obrázku 8.5.

Z dat nevyplývá lepší generická pozice pro tuto dekompozici. Nelze říci, že určitá pozice na mapě, ať už ve středu či u nějakého kraje, by byla lepší než jiná. Můžeme však vidět, že délku trasy ovlivňuje množství úzkých buněk, které pak generují překrývající se cesty efektoru.

Kapitola 9

Závěr

V rámci této práce byly studovány algoritmy plánování cesty ve známém i neznámém prostoru. Celkově bylo implementováno devět algoritmů. Pět z nich s účelem navigace a čtyři algoritmy vytvářející dekompozice umožňující pokrytí prostoru cestou.

Dále byla v Javě vytvořena demonstrační aplikace, která umožňuje krokování těchto algoritmů a jejich vzájemné porovnání. Byla využita v této práci pro porovnání statistik z jednotlivých běhů.

Ze studie vyplývá výkonnostní převaha Tangent Bugu coby navigačního algoritmu v neznámém prostředí. Ve známém prostředí pak dominují Potenciálová pole, které díky své robustnosti předčí ostatní. V rámci testování buněčných dekompozicí byla potvrzena teorie o vztahu Lichoběžníkových a Boustrophedon dekompozicí, protože v každém představeném případě druhá zmíněná generovala lepší výsledky. Kruhové a Brushfire dekompozice předčily pouze ve složitosti implementace, avšak experimentální výsledky neukázaly žádné výhody. Dokonce na mapách s malým množství komplexních překážek se neukázaly jako vhodné. To je způsobeno vytvářením velkého množství úzkých buněk, které pro své pokrytí vyžadují minimálně jednu cestu po řezu.

V rámci studia popisovaných přístupů jsou časté příklady prostředí s vnější překážkou, tedy překážkou omezující vnější hranici dostupného prostoru. Takové překážky knihovna *vizlib* neumožňuje vytvářet a rozšíření to není triviální. Dalším postupem v této práci by bylo zvýšení robustnosti implementovaných algoritmů a rozšíření knihovny o zmíněné vnější překážky. Demonstrační program by měl být dále rozšířen o implementace algoritmů od jiných autorů, aby bylo možné vytvořit komplexní, komparativní studii.

Literatura

[1] Canny, J. : Constructing roadmaps of semi-algebraic sets I: Completeness. *Artificial Intelligence*, roč. 37, č. 1, 1988: s 203 – 222, ISSN 0004-3702, https://doi.org/10.1016/0004-3702(88)90055-0. Dostupné z: <http://www.sciencedirect.com/science/article/pii/0004370288900550>

[2] Choset, H. M. *Principles of robot motion: theory, algorithms, and implementation.* MIT Press, c2005. ISBN 0-262-03327-5.

[3] Hart, P. E.; Nilsson, N. J.; Raphael, B. : A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, roč. 4, č. 2, July 1968: s 100–107, ISSN 0536-1567, 10.1109/TSSC.1968.300136.

[4] JetBrains : IntelliJ IDEA. [Online; navštíveno 1.1.2017]. Dostupné z: <https://www.jetbrains.com/idea/>

[5] Rusnák, J. *Vizualizace algoritmů pro plánování cesty* 2017. Dostupné z: <http://www.fit.vutbr.cz/study/DP/DP.php?id=18368>.

[6] Schwartz, J. T.; Sharir, M. : On the "piano movers" problem. II. General techniques for computing topological properties of real algebraic manifolds. *Advances in Applied Mathematics*, roč. 4, č. 3, 1983: s 298 – 351, ISSN 0196-8858, https://doi.org/10.1016/0196-8858(83)90014-3. Dostupné z: <http://www.sciencedirect.com/science/article/pii/0196885883900143>

[7] Wikipedia : Motion planning. [Online; navštíveno 1.1.2017]. Dostupné z: <https://en.wikipedia.org/wiki/Motion_planning>

Příloha A

Obsah paměťového média

- `app` – složka demonstrační aplikace
  - `PathPlanning.jar` – binární soubor aplikace
  - `src` – složka obsahující zdrojové soubory aplikace
- `text` – složka dokumentace
  - `DP_xrepka04.pdf` – soubor dokumentace
  - `DP_xrepka04_print.pdf` – soubor dokumentace k tisku
  - `src` – složka obsahující zdrojové soubory dokumentace

Příloha B

Úprava a přeložení aplikace

Pro spuštění aplikace je vyžadována Java v aktuální verzi.

Pokud je vyžadována úprava a přeložení ze zdrojových souborů, je možné pomocí *IntelliJ* importovat existující projekt a vytvořit nový build. V tomto případě je nutné nainstalovat také knihovnu *vizlib*. Její instalační proces je zevrubně popsán v přílohách práce Jakuba Rusnáka [5]. V krátkosti je tedy postup následovný:

- instalace IntelliJ [4]
- import existujícího projektu (cíl pro Maven `pom.xml`) [5]
- instalace knihovny *vizlib* [5]
- vytvoření konfigurace běhu *run* programu [5]
- spuštění